

Verifying Monadic Second-Order Properties of Graph Programs

— *extended version* —

Christopher M. Poskitt¹ and Detlef Plump²

¹ Department of Computer Science, ETH Zürich, Switzerland

² Department of Computer Science, The University of York, UK

Updated: 16th June 2014

Abstract. The core challenge in a Hoare- or Dijkstra-style proof system for graph programs is in defining a weakest liberal precondition construction with respect to a rule and a postcondition. Previous work addressing this has focused on assertion languages for first-order properties, which are unable to express important global properties of graphs such as acyclicity, connectedness, or existence of paths. In this paper, we extend the nested graph conditions of Habel, Pennemann, and Rensink to make them equivalently expressive to monadic second-order logic on graphs. We present a weakest liberal precondition construction for these assertions, and demonstrate its use in verifying non-local correctness specifications of graph programs in the sense of Habel et al.

1 Introduction

Many problems in computer science and software engineering can be modelled in terms of graphs and graph transformation, including the specification and analysis of pointer structures, object-oriented systems, and model transformations; to name just a few. These applications, amongst others, motivate the development of techniques for verifying the functional correctness of both graph transformation rules and programs constructed over them.

A recent strand of research along these lines has resulted in the development of *proof calculi* for graph programs. These, in general, provide a means of systematically proving that a program is correct relative to a specification. A first approach was considered by Habel, Pennemann, and Rensink [8,14], who contributed weakest precondition calculi – in the style of Dijkstra – for simple rule-based programs, with specifications expressed using *nested conditions* [7]. Subsequently, we developed Hoare logics [18,17] for the graph transformation language GP 2 [16], which additionally allows computation over labels, and employed as a specification language an extension of nested conditions with support for expressions.

Both approaches suffer from a common drawback, in that they are limited to first-order structural properties. In particular, neither of them support proofs

about important *non-local* properties of graphs, e.g. acyclicity, connectedness, or the existence of arbitrary-length paths. Part of the difficulty in supporting such assertions is at the core of both approaches: defining an effective construction for the weakest property guaranteeing that an application of a given rule will establish a given postcondition (i.e. the construction of a *weakest liberal precondition* for graph transformation rules).

Our paper addresses exactly this challenge. We define an extension of nested conditions that is equivalently expressive to monadic second-order (MSO) logic on graphs [3]. For this assertion language, and for graph programs similar to those of [8,14], we define a weakest liberal precondition construction that can be integrated into Dijkstra- and Hoare-style proof calculi. Finally we demonstrate its use in verifying non-local correctness specifications (properties including that the graph is bipartite, acyclic) of some simple programs.

The paper is organised as follows. In Section 2 we provide some preliminary definitions and notations. In Section 3 we define an extension of nested conditions for MSO properties. In Section 4 we define graph programs, before presenting our weakest liberal precondition construction in Section 5, and demonstrating in Section 6 its use in Hoare-style correctness proofs. Finally, Section 7 presents some related work before we conclude the paper in Section 8.

This is an extended version of [20], and includes the semantics of graph programs as well as the missing proofs.

2 Preliminaries

Let $\mathbb{B} = \{\text{true}, \text{false}\}$ denote the set of Boolean values, Vertex, Edge denote (disjoint) sets of node and edge identifiers (which shall be written in lowercase typewriter font, e.g. v, e), and VSetVar, ESetVar denote (disjoint) sets of node- and edge-set variables (which shall be written in uppercase typewriter font, e.g. X, Y).

A *graph* over a label alphabet $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ is defined as a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where $V_G \subset \text{Vertex}$ and $E_G \subset \text{Edge}$ are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G: E_G \rightarrow V_G$ are the *source* and *target* functions for edges, $l_G: V_G \rightarrow \mathcal{C}_V$ is the node labelling function and $m_G: E_G \rightarrow \mathcal{C}_E$ is the edge labelling function. The *empty graph*, denoted by \emptyset , has empty node and edge sets. For simplicity, we fix the label alphabet throughout this paper as $\mathcal{L} = \langle \{\square\}, \{\square\} \rangle$, where \square denotes the blank label (which we render as \bullet and \longrightarrow in pictures). We note that our technical results hold for any fixed finite label alphabet.

Given a graph G , the (*directed*) *path predicate* $\text{path}_G: V_G \times V_G \times 2^{E_G} \rightarrow \mathbb{B}$ is defined inductively for nodes $v, w \in V_G$ and sets of edges $E \subseteq E_G$. If $v = w$, then $\text{path}_G(v, w, E)$ holds. If $v \neq w$, then $\text{path}_G(v, w, E)$ holds if there exists an edge $e \in E_G \setminus E$ such that $s_G(e) = v$ and $\text{path}_G(t_G(e), w, E)$.

A *graph morphism* $g: G \rightarrow H$ between graphs G, H consists of two functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources, targets and labels; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $l_H \circ g_V = l_G$, and $m_H \circ g_E = m_G$.

We call G, H the *domain* (resp. *codomain*) of g . Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x . It is *injective* (*surjective*) if g_V and g_E are injective (surjective). It is an *isomorphism* if it is both injective and surjective. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

3 Expressing Monadic Second-Order Properties

We extend the nested conditions of [7] to a formalism equivalently expressive to MSO logic on graphs. The idea is to introduce new quantifiers for node- and edge-set variables, and equip morphisms with constraints about set membership. The definition of satisfaction is then extended to require an interpretation of these variables in the graph such that the constraint evaluates to true. Furthermore, constraints can also make use of a predicate for explicitly expressing properties about directed paths. Such properties can of course be expressed in terms of MSO expressions, but the predicate is provided as a more compact alternative.

Definition 1 (Interpretation; interpretation constraint). Given a graph G , an *interpretation* I in G is a partial function $I : \text{VSetVar} \cup \text{ESetVar} \rightarrow 2^{V_G} \cup 2^{E_G}$, such that for all variables \mathbf{X} on which it is defined, $I(\mathbf{X}) \in 2^{V_G}$ if $\mathbf{X} \in \text{VSetVar}$ (resp. 2^{E_G} , ESetVar). An (*interpretation*) *constraint* is a Boolean expression that can be derived from the syntactic category `Constraint` of the following grammar:

$$\begin{aligned} \text{Constraint} ::= & \text{Vertex } ' \in ' \text{ VSetVar} \mid \text{Edge } ' \in ' \text{ ESetVar} \\ & \mid \text{path } ' (' \text{ Vertex } ', ' \text{ Vertex } [', ' \text{ not Edge } \{ ' | ' \text{ Edge } \}] ') ' \\ & \mid \text{not Constraint} \mid \text{Constraint (and} \mid \text{or) Constraint} \mid \text{true} \end{aligned}$$

Given a constraint γ , an interpretation I in G , and a morphism q with codomain G , the value of $\gamma^{I,q}$ in \mathbb{B} is defined inductively. If γ contains a set variable for which I is undefined, then $\gamma^{I,q} = \text{false}$. Otherwise, if γ is `true`, then $\gamma^{I,q} = \text{true}$. If γ has the form $x \in \mathbf{X}$ with x a node or edge identifier and \mathbf{X} a set variable, then $\gamma^{I,q} = \text{true}$ if $q(x) \in I(\mathbf{X})$. If γ has the form `path(v, w)` with v, w node identifiers, then $\gamma^{I,q} = \text{true}$ if the predicate $\text{path}_G(q(v), q(w), \emptyset)$ holds. If γ has the form `path(v, w , not $e_1 \mid \dots \mid e_n$)` with v, w node identifiers and e_1, \dots, e_n edge identifiers, then $\gamma^{I,q} = \text{true}$ if it is the case that the path predicate $\text{path}_G(q(v), q(w), \{q(e_1), \dots, q(e_n)\})$ holds. If γ has the form `not γ_1` with γ_1 a constraint, then $\gamma^{I,q} = \text{true}$ if $\gamma_1^{I,q} = \text{false}$. If γ has the form `γ_1 and γ_2` (resp. `γ_1 or γ_2`) with γ_1, γ_2 constraints, then $\gamma^{I,q} = \text{true}$ if both (resp. at least one of) $\gamma_1^{I,q}$ and $\gamma_2^{I,q}$ evaluate(s) to true. \square

Definition 2 (M-condition; M-constraint). An *MSO condition* (short. *M-condition*) over a graph P is of the form `true`, $\exists_V \mathbf{X}[c]$, $\exists_E \mathbf{X}[c]$, or $\exists(a \mid \gamma, c')$, where $\mathbf{X} \in \text{VSetVar}$ (resp. ESetVar), c is an M-condition over P , $a : P \hookrightarrow C$ is an injective morphism (since we consider programs with injective matching), γ is an interpretation constraint over items in C , and c' is an M-condition over C . Furthermore, Boolean formulae over M-conditions over P are also M-conditions

over P ; that is, $\neg c$, $c_1 \wedge c_2$, and $c_1 \vee c_2$ are M-conditions over P if c, c_1, c_2 are M-conditions over P .

An M-condition over the empty graph \emptyset in which all set variables are bound to quantifiers is called an *M-constraint*. \square

For brevity, we write **false** for $\neg \mathbf{true}$, $c \Rightarrow d$ for $\neg c \vee d$, $c \Leftrightarrow d$ for $c \Rightarrow d \wedge d \Rightarrow c$, $\forall_v \mathbf{X}[c]$ for $\neg \exists_v \mathbf{X}[\neg c]$, $\forall_E \mathbf{X}[c]$ for $\neg \exists_E \mathbf{X}[\neg c]$, $\exists_v \mathbf{X}_1, \dots, \mathbf{X}_n[c]$ for $\exists_v \mathbf{X}_1[\dots \exists_v \mathbf{X}_n[c] \dots]$ (analogous for other set quantifiers), $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \mathbf{true})$, $\exists(a, c')$ for $\exists(a \mid \mathbf{true}, c')$, and $\forall(a \mid \gamma, c')$ for $\neg \exists(a \mid \gamma, \neg c')$.

In our examples, when the domain of a morphism $a: P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain C . For instance, an M-constraint $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$.

Definition 3 (Satisfaction of M-conditions). Let $p: P \hookrightarrow G$ denote an injective morphism, c an M-condition over P , and I an interpretation in G . We define inductively the meaning of $p \models^I c$, which denotes that p satisfies c with respect to I . If c has the form **true**, then $p \models^I c$. If c has the form $\exists_v \mathbf{X}[c']$ (resp. $\exists_E \mathbf{X}[c']$), then $p \models^I c$ if $p \models^{I'} c'$, where $I' = I \cup \{\mathbf{X} \mapsto V\}$ for some $V \subseteq V_G$ (resp. $\{\mathbf{X} \mapsto E\}$ for some $E \subseteq E_G$). If c has the form $\exists(a: P \hookrightarrow C \mid \gamma, c')$, then $p \models^I c$ if there is an injective morphism $q: C \hookrightarrow G$ such that $q \circ a = p$, $\gamma^{I \cdot q} = \mathbf{true}$, and $q \models^I c'$.

A graph G satisfies an M-constraint c , denoted $G \models c$, if $i_G: \emptyset \hookrightarrow G \models^{I_\emptyset} c$, where I_\emptyset is the *empty interpretation in G* , i.e. undefined on all set variables. \square

We remark that model checking for both first-order and monadic second-order logic is known to be PSPACE-complete [5]. However, the model checking problem for monadic second-order logic on graphs of bounded treewidth can be solved in linear time [2].

Example 1. The following M-constraint *col* (translated from the corresponding formula §1.5 of [1]) expresses that a graph is 2-colourable (or bipartite); i.e. every node can be assigned one of two colours such that no two adjacent nodes have the same one. Let γ_{col} denote **not** ($v \in \mathbf{X}$ and $w \in \mathbf{X}$) and **not** ($v \in \mathbf{Y}$ and $w \in \mathbf{Y}$).

$$\begin{aligned} \exists_v \mathbf{X}, \mathbf{Y} [\quad & \forall(\bullet_v, \exists(\bullet_v \mid (v \in \mathbf{X} \text{ or } v \in \mathbf{Y}) \text{ and not } (v \in \mathbf{X} \text{ and } v \in \mathbf{Y}))) \\ & \wedge \forall(\bullet_v \bullet_w, \exists(\bullet_v \xrightarrow{\quad} \bullet_w) \Rightarrow \exists(\bullet_v \bullet_w \mid \gamma_{\text{col}})) \quad] \end{aligned}$$

A graph G will satisfy *col* if there exist two subsets of V_G such that: (1) every node in G belongs to *exactly one* of the two sets; and (2) if there is an edge from one node to another, then those nodes are not in the same set. Intuitively, one can think of the sets \mathbf{X} and \mathbf{Y} as respectively denoting the nodes of colour one and colour two. If two such sets do not exist, then the graph cannot be assigned a 2-colouring. \square

Theorem 1 (M-constraints are equivalent to MSO formulae). The assertion languages of M-constraints and MSO graph formulae are equivalently expressive: that is, given an M-constraint c , there exists an MSO graph formula φ such that for all graphs G , $G \models c$ if and only if $G \models \varphi$; and vice versa. \square

Proof. See Appendix A.

4 Graph Programs

In this section we define rules, rule application, and graph programs. Whilst the syntax and semantics of the control constructs are based on those of GP 2 [16], the rules themselves follow [8,14], i.e. are labelled over a fixed finite alphabet, and do not support relabelling or expressions. We equip the rules with application conditions (M-conditions over the left- and right-hand graphs), and define *rule application* via the standard double-pushout construction [4].

Definition 4 (Rule; direct derivation). A *plain rule* $r' = \langle L \leftarrow K \hookrightarrow R \rangle$ comprises two inclusions $K \hookrightarrow L$, $K \hookrightarrow R$. We call L, R the left- (resp. right-) hand graph and K the interface. An *application condition* $\text{ac} = \langle \text{ac}_L, \text{ac}_R \rangle$ for r' consists of two M-conditions over L and R respectively. A *rule* $r = \langle r', \text{ac} \rangle$ is a plain rule r' and an application condition ac for r' .

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ g \downarrow & (1) & \downarrow & (2) & \downarrow h \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

For a plain rule r' and a morphism $K \hookrightarrow D$, a *direct derivation* $G \Rightarrow_{r',g,h} H$ (short. $G \Rightarrow_{r'} H$ or $G \Rightarrow H$) is given by the pushouts (1) and (2). For a rule $r = \langle r', \text{ac} \rangle$, there is a *direct derivation* $G \Rightarrow_{r,g,h} H$ if $G \Rightarrow_{r',g,h} H$, $g \models^{I_0} \text{ac}_L$, and $h \models^{I_0} \text{ac}_R$. We call g, h a *match* (resp. *comatch*) for r . Given a set of rules \mathcal{R} , we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_{r,g,h} H$ for some $r \in \mathcal{R}$. \square

It is known that, given a (plain) rule r , graph G , and morphism g as above, there exists a direct derivation if and only if g satisfies the *dangling condition*, i.e. that no node in $g(L) \setminus g(K)$ is incident to an edge in $G \setminus g(L)$. In this case, D and H are determined uniquely up to isomorphism, constructed from G as follows: first, remove all edges in $g(L) \setminus g(K)$ obtaining D . Then add disjointly all nodes and edges from $R \setminus K$ retaining their labels. For $e \in E_R \setminus E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R \setminus V_K$, otherwise $s_H(e) = g_V(s_R(e))$, (targets defined analogously) resulting in the graph H .

We will often give rules without the interface, writing just $L \Rightarrow R$. In such cases we number nodes that correspond in L and R , and establish the convention that K comprises exactly these nodes and that $E_K = \emptyset$ (i.e. K can be completely inferred from L, R). Furthermore, if the application condition of a rule is $\langle \text{true}, \text{true} \rangle$, then we will only write the plain rule component.

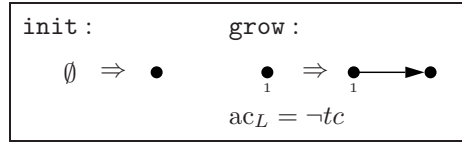
We consider now the syntax and semantics of graph programs, which provide a mechanism to control the application of rules to some graph provided as input.

Definition 5 (Graph program). (*Graph*) *programs* are defined inductively. First, every rule (resp. rule set) r, \mathcal{R} and **skip** are programs. Given programs C, P, Q , we have that $P; Q$, $P!$, if C then P else Q , and try C then P else Q are programs. \square

Graph programs are *nondeterministic*, and their execution on a particular graph could result in one of several possible outcomes. That outcome could be a graph, or it could be the special state “fail” which occurs when a rule (set) is not *applicable* to the current graph.

An operational semantics for programs is given in Appendix B, but the informal meaning of the constructs is as follows. Let G denote an input graph. Programs r, \mathcal{R} correspond to rule (resp. rule set) application, returning H if there exists some $G \Rightarrow_r H$ (resp. $G \Rightarrow_{\mathcal{R}} H$); otherwise fail. Program $P; Q$ denotes sequential composition. Program $P!$ denotes as-long-as-possible iteration of P . Finally, the conditional programs execute the first or second branch depending on whether executing C returns a graph or fail, with the distinction that the **if** construct does not retain any effects of C , whereas the **try** construct does.

Example 2. Consider the program **init**; **grow**! defined by the rules:



where tc is an (unspecified) M-condition over L expressing some termination condition for the iteration (proving termination is not our concern here, see e.g. [19]). The program, if executed on the empty graph, nondeterministically constructs and returns a tree. It applies the rule **init** exactly once, creating an isolated node. It then iteratively applies the rule **grow** (each application adding a leaf to the tree) until the termination condition tc holds. An example program run, with $tc = \exists(\bullet_1 \bullet \bullet \bullet)$, is:



□

5 Constructing a Weakest Liberal Precondition

In this section, we present a construction for the *weakest liberal precondition* relative to a rule r and a postcondition c (which is an M-constraint). In our terminology, if a graph satisfies a weakest liberal precondition, then: (1) any graphs resulting from applications of r will satisfy c ; and (2) there does not exist another M-constraint with this property that is weaker. (Note that we do not address termination or existence of results in this paper.)

The construction is adapted from the one for nested conditions in [7], and as before, is broken down into a number of stages. First, a translation of postconditions into M-conditions over R (transformation “A”); then, from M-conditions over R into M-conditions over L (transformation “L”); and finally, from M-conditions over L into an M-constraint expressing the weakest liberal precondition (via transformations “App” and “Pre”).

First, we consider transformation A , which constructs an M-condition over R from a postcondition (an M-constraint) by computing a disjunction over all the ways that the M-constraint and comatches might “overlap”.

Theorem 2 (M-constraints to M-conditions over R). There is a transformation A , such that for all M-constraints c , all rules r with right-hand side R , and all injective morphisms $h: R \hookrightarrow H$,

$$h \models^{I_\emptyset} A(r, c) \text{ if and only if } H \models c.$$

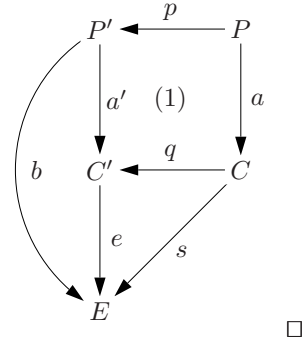
Construction. Let c denote an M-constraint, and r a rule with right-hand side R . We define $A(r, c) = A'(\emptyset \hookrightarrow R, c)$ where A' is defined inductively as follows. For injective graph morphisms $p: P \hookrightarrow P'$ and M-conditions over P , define:

$$\begin{aligned} A'(p, \text{true}) &= \text{true}, \\ A'(p, \exists_v X[c']) &= \exists_v X[A'(p, c')], \\ A'(p, \exists_E X[c']) &= \exists_E X[A'(p, c')], \\ A'(p, \exists(a: P \hookrightarrow C \mid \gamma, c')) &= \bigvee_{e \in \varepsilon} \exists(b: P' \hookrightarrow E \mid \gamma, A'(s: C \hookrightarrow E, c')). \end{aligned}$$

The final equation relies on the following. First, construct the pushout (1) of p and a leading to injective graph morphisms $a': P' \hookrightarrow C'$ and $q: C \hookrightarrow C'$.

The disjunction then ranges over the set ε , which we define to contain every surjective graph morphism $e: C' \rightarrow E$ such that $b = e \circ a'$ and $s = e \circ q$ are injective graph morphisms (we consider the codomains of each e up to isomorphism, hence the disjunction is finite).

The transformations A, A' are extended for Boolean formulae over M-conditions in the usual way, that is, $A(r, \neg c) = \neg A(r, c)$, $A(r, c_1 \wedge c_2) = A(r, c_1) \wedge A(r, c_2)$, and $A(r, c_1 \vee c_2) = A(r, c_1) \vee A(r, c_2)$ (analogous for A').



Example 3. Recall the rule **grow** from Example 2. Let c denote the M-constraint:

$$\exists_v X, Y [\forall (\bullet_v \bullet_w, \exists (\bullet_v \bullet_w \mid \text{path}(v, w)) \Rightarrow \exists (\bullet_v \bullet_w \mid \gamma))]$$

for $\gamma = (v \in X \text{ and } w \in Y) \text{ and not } (v \in Y \text{ or } w \in X)$, which expresses that there are two sets of nodes X, Y in the graph, such that if there is a path from some node v to some node w , then v belongs only to X and w only to Y . Applying transformation A :

$$\begin{aligned}
& A(\text{grow}, c) \\
&= A'(\emptyset \hookrightarrow \bullet_1 \rightarrow \bullet_2, c) \\
&= \exists_v X, Y [A'(\emptyset \hookrightarrow \bullet_1 \rightarrow \bullet_2, \forall(\bullet_v \bullet_w, \\
&\quad \exists(\bullet_v \bullet_w \mid \text{path}(v, w)) \Rightarrow \exists(\bullet_v \bullet_w \mid \gamma)))] \\
&= \exists_v X, Y [\bigwedge_{i=1}^7 \forall(\bullet_1 \rightarrow \bullet_2 \hookrightarrow E_i, \exists(E_i \mid \text{path}(v, w)) \Rightarrow \exists(E_i \mid \gamma))]
\end{aligned}$$

where the graphs E_i are as given in Figure 1. □

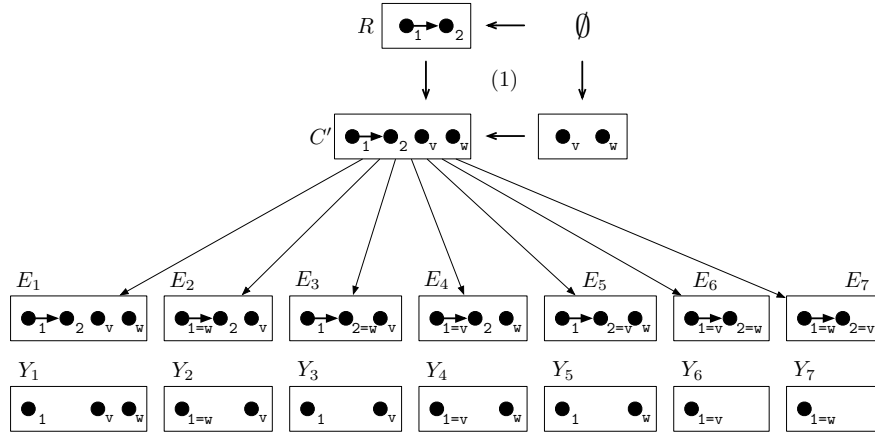


Fig. 1. Applying the construction in Examples 3 and 4

In order to prove the statement of Theorem 4, we first prove a more general lemma stating that an M-condition over P can be shifted over a morphism p with domain P . It is a generalised version of Lemma 3 in [7], but the proof is almost identical as interpretation constraints are not manipulated by this transformation, and both sides of the statement are interpreted in the same graph.

Lemma 1 (Shifting M-conditions over morphisms). For all M-conditions c over P , all interpretations I in H , and all injective morphisms $p: P \hookrightarrow P', p'': P' \hookrightarrow H$, we have:

$$p'' \models^I A'(p, c) \text{ if and only if } p'' \circ p \models^I c.$$

□

Proof. See Appendix C.1. □

Theorem 4 then follows as an instance of Lemma 1.

Proof (of Theorem 4). With the construction of A , Lemma 1, and the definition of \models , we have: $h \models^{I_0} A(r, c)$ iff $h \models^{I_0} A'(i_R : \emptyset \hookrightarrow R, c)$ iff $h \circ i_R \models^{I_0} c$ iff $i_H : \emptyset \hookrightarrow H \models^{I_0} c$ iff $H \models c$. \square

Transformation L , adapted from [7], takes an M -condition over R and constructs an M -condition over L that is satisfied by a match if and only if the original is satisfied by the comatch. The transformation is made more complex by the presence of path and MSO expressions, because nodes and edges referred to on the right-hand side may no longer exist on the left. For clarity, we separate the handling of these two types of expressions, and in particular, define a *decomposition* $LPath$ of path predicates according to the items that the rule is creating or deleting. For example, if an edge is created by a rule, a path predicate decomposes to a disjunction of path predicates collectively asserting the existence of paths to and from the nodes that will eventually become its source and target; whereas if an edge is to be deleted, the predicate will exclude it.

Proposition 1 (Path decomposition). There is a transformation $LPath$ such that for every rule $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$, direct derivation $G \Rightarrow_{r,g,h} H$, path predicate p over R , and interpretation I ,

$$LPath(r, p)^{I,g} = p^{I,h}.$$

Construction. Let $r = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and $p = \text{path}(v, w, \text{not } E)$. For simplicity, we will treat the syntactic construct E as a set of edges and identify $\text{path}(v, w, \text{not } E)$ and $\text{path}(v, w)$ when E is empty. Then, define:

$$LPath(r, p) = LPath'(r, v, w, E^\ominus) \text{ or } \text{FuturePaths}(r, p).$$

Here, E^\ominus is constructed from E by adding edges $e \in E_L \setminus E_R$, i.e. that the rule will delete. Furthermore, $LPath'(r, v, w, E^\ominus)$ decomposes to path predicates according to whether v and w exist in K . If $\text{path}_R(v, w, E^\ominus)$ holds, then $LPath'(r, v, w, E^\ominus)$ returns **true**. Otherwise, if both $v, w \in V_K$, then it returns $\text{path}(v, w, \text{not } E^\ominus)$. If $v \notin V_K, w \in V_K$, it returns:

$$\text{false or path}(x_1, w, \text{not } E^\ominus) \text{ or path}(x_2, w, \text{not } E^\ominus) \text{ or } \dots$$

for each $x_i \in V_K$ such that $\text{path}_R(v, x_i, E^\ominus)$. Case $v \in V_K, w \notin V_K$ analogous. If $v, w \notin V_K$, then it returns **false or path** $(x_i, y_j, \text{not } E^\ominus)$ **or** ... for all $x_i, y_j \in V_K$ such that $\text{path}_R(v, x_i, E^\ominus)$ and $\text{path}_R(y_j, w, E^\ominus)$.

Finally, $\text{FuturePaths}(r, p)$ denotes **false** in disjunction with:

$$(\text{LPath}'(r, v, x_1, E^\ominus) \text{ and path}(y_1, x_2, \text{not } E^\ominus) \dots \text{and path}(y_i, x_{i+1}, \text{not } E^\ominus) \dots \text{and LPath}'(r, y_n, w, E^\ominus))$$

over all non-empty sequences of distinct pairs $\langle \langle x_1, y_1 \rangle, \dots, \langle x_n, y_n \rangle \rangle$ drawn from:

$$\{ \langle x, y \rangle \mid x, y \in V_K \wedge \text{path}_R(x, y, E^\ominus) \wedge \neg \text{path}_L(x, y, E^\ominus) \}.$$

\square

Proof. See Section C.2. □

In addition to paths, transformation L must handle MSO expressions that refer to items present in R but absent in L . To achieve this, it computes a disjunction over all possible “future” (i.e. immediately after the rule application) set memberships of these missing items. The idea being, that if a set membership exists for these missing items that satisfies the interpretation constraints *before* the rule application, then one will still exist once they have been created. The transformation keeps track of such potential memberships via sets of pairs as follows.

Definition 6 (Membership set). A *membership set* M is a set of pairs (x, \mathbf{X}) of node or edge identifiers x with set variables of the corresponding type. Intuitively, $(x, \mathbf{X}) \in M$ encodes that $x \in \mathbf{X}$, whereas $(x, \mathbf{X}) \notin M$ encodes that $x \notin \mathbf{X}$. □

Theorem 3 (From M-conditions over R to L). There is a transformation L such that for every rule $r = \langle \langle L \leftrightarrow K \hookrightarrow R \rangle, \text{ac} \rangle$, every M-condition c over R (with no free variables, and distinct variables for distinct quantifiers), and every direct derivation $G \Rightarrow_{r,g,h} H$,

$$g \models^{I_\emptyset} L(r, c) \text{ if and only if } h \models^{I_\emptyset} c.$$

Construction. Let $r = \langle \langle L \leftrightarrow K \hookrightarrow R \rangle, \text{ac} \rangle$ denote a rule and c an M-condition over R . We define $L(r, c) = L'(r, c, \emptyset)$. For such an r, c , and membership set M , the transformation L' is defined inductively as follows:

$$\begin{aligned} L'(r, \text{true}, M) &= \text{true}, \\ L'(r, \exists_V \mathbf{X}[c'], M) &= \exists_V \mathbf{X} \left[\bigvee_{M' \in 2^{M_V}} L'(r, c', M \cup M') \right] \\ L'(r, \exists_E \mathbf{X}[c'], M) &= \exists_E \mathbf{X} \left[\bigvee_{M' \in 2^{M_E}} L'(r, c', M \cup M') \right] \end{aligned}$$

where $M_V = \{(v, \mathbf{X}) \mid v \in V_R \setminus V_L\}$ and $M_E = \{(e, \mathbf{X}) \mid e \in E_R \setminus E_L\}$.

For case $c = \exists(a \mid \gamma, c')$, we define:

$$L'(r, \exists(a \mid \gamma, c'), M) = \text{false}$$

if $\langle K \hookrightarrow R, a \rangle$ has no pushout complement; otherwise:

$$L'(r, \exists(a \mid \gamma, c'), M) = \exists(b \mid \gamma_M, L'(r^*, c', M))$$

which relies on the following. First, construct the pushout (1), with $r^* = \langle Y \leftarrow Z \hookrightarrow X \rangle$ the “derived” rule obtained by constructing pushout (2). The interpretation constraint γ_M is obtained from γ as follows. First, consider each predicate $x \in \mathbf{X}$ such that $x \notin Y$. If $(y, \mathbf{X}) \in M$ for some $y = x$, replace the predicate with **true**; otherwise **false**. Then, replace each path predicate p with $\text{LPath}(r^*, p)$.

$$\begin{array}{ccccc} r: \langle L & \xleftarrow{\quad} & K & \xrightarrow{\quad} & R \rangle \\ & \downarrow b & & \downarrow & \downarrow a \\ & & (2) & & (1) \\ r^*: \langle Y & \xleftarrow{\quad} & Z & \xrightarrow{\quad} & X \rangle \end{array}$$

The transformation L is extended for Boolean formulae in the usual way, that is, $L(r, \neg c) = \neg L(r, c)$, $L(r, c_1 \wedge c_2) = L(r, c_1) \wedge L(r, c_2)$, and $L(r, c_1 \vee c_2) = L(r, c_1) \vee L(r, c_2)$ (analogous for L'). \square

Example 4. Take **grow**, c , γ and $A(\text{grow}, c)$ as considered in Example 3. Applying transformation L :

$$\begin{aligned} L(\text{grow}, A(\text{grow}, c)) &= L'(\text{grow}, A(\text{grow}, c), \emptyset) \\ &= \exists \mathbf{v} \mathbf{X}, \mathbf{Y} [\bigvee_{M' \in 2^{M_V}} L'(\text{grow}, \bigwedge_{i=1}^7 (\bullet_1 \xrightarrow{\quad} \bullet_2 \hookrightarrow E_i, \exists(E_i \mid \text{path}(\mathbf{v}, \mathbf{w})) \\ &\quad \Rightarrow \exists(E_i \mid \gamma)), M')] \\ &= \exists \mathbf{v} \mathbf{X}, \mathbf{Y} [\bigvee_{M' \in 2^{M_V}} (\bigwedge_{i \in \{1,2,4\}} \forall(\bullet_1 \hookrightarrow Y_i, \exists(Y_i \mid \text{path}(\mathbf{v}, \mathbf{w})) \Rightarrow \exists(Y_i \mid \gamma)) \\ &\quad \wedge \forall(\bullet_1 \bullet_{\mathbf{v}}, \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \text{path}(\mathbf{v}, 1)) \Rightarrow \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \gamma_{M'}, L'(\text{grow}, \text{true}, M'))) \\ &\quad \wedge \forall(\bullet_1 \bullet_{\mathbf{w}}, \text{false} \Rightarrow \exists(\bullet_1 \bullet_{\mathbf{w}} \mid \gamma_{M'}, L'(\text{grow}, \text{true}, M'))) \\ &\quad \wedge \forall(\bullet_{1=\mathbf{v}}, \text{true} \Rightarrow \exists(\bullet_{1=\mathbf{v}} \mid \gamma_{M'}, L'(\text{grow}, \text{true}, M'))) \\ &\quad \wedge \forall(\bullet_{1=\mathbf{w}}, \text{false} \Rightarrow \exists(\bullet_{1=\mathbf{w}} \mid \gamma_{M'}, L'(\text{grow}, \text{true}, M'))))] \\ &= \exists \mathbf{v} \mathbf{X}, \mathbf{Y} [\bigvee_{M' \in 2^{M_V}} (\bigwedge_{i \in \{1,2,4\}} \forall(\bullet_1 \hookrightarrow Y_i, \exists(Y_i \mid \text{path}(\mathbf{v}, \mathbf{w})) \Rightarrow \exists(Y_i \mid \gamma)) \\ &\quad \wedge \forall(\bullet_1 \bullet_{\mathbf{v}}, \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \text{path}(\mathbf{v}, 1)) \Rightarrow \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \gamma_{M'})) \\ &\quad \wedge \forall(\bullet_{1=\mathbf{v}}, \exists(\bullet_{1=\mathbf{v}} \mid \gamma_{M'})))] \\ &= \exists \mathbf{v} \mathbf{X}, \mathbf{Y} [\bigwedge_{i \in \{1,2,4\}} \forall(\bullet_1 \hookrightarrow Y_i, \exists(Y_i \mid \text{path}(\mathbf{v}, \mathbf{w})) \Rightarrow \exists(Y_i \mid \gamma)) \\ &\quad \wedge \forall(\bullet_1 \bullet_{\mathbf{v}}, \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \text{path}(\mathbf{v}, 1)) \Rightarrow \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \mathbf{v} \in \mathbf{X} \text{ and not } \mathbf{v} \in \mathbf{Y})) \\ &\quad \wedge \forall(\bullet_{1=\mathbf{v}}, \exists(\bullet_{1=\mathbf{v}} \mid \mathbf{v} \in \mathbf{X} \text{ and not } \mathbf{v} \in \mathbf{Y}))] \end{aligned}$$

where the graphs E_i and Y_i are as given in Figure 1 and $M_V = \{(2, \mathbf{X}), (2, \mathbf{Y})\}$. Here, only one of the subsets ranged over yields a satisfiable disjunct: $M' = \{(2, \mathbf{Y})\}$, i.e. $\gamma_{M'} = (\mathbf{v} \in \mathbf{X} \text{ and true})$ and not $(\mathbf{v} \in \mathbf{Y} \text{ or false})$ for $\mathbf{w} = 2$. \square

In order to prove the statement about L (which is interpreted over I_\emptyset), we need to prove a more general lemma.

Lemma 2. There is a transformation L such that for every rule $r = \langle \langle L \leftarrow K \hookrightarrow R \rangle, ac \rangle$, every M -condition c over R with distinct variables for distinct quantifiers, every interpretation I in G defined for all free set variables of c , every membership set M such that $(x, _) \in M$ implies $x \in R \setminus L$, and every direct derivation $G \Rightarrow_{r,g,h} H$,

$$g \models^I L'(r, c, M) \text{ if and only if } h \models^{I_M} c.$$

Here, I_M is defined as I except for all $x \in R \setminus L$, where $h(x) \in I_M(\mathbf{x})$ if and only if $(x, \mathbf{x}) \in M$. \square

Proof. See Appendix C.3. \square

Proof (of Theorem 4). With the construction of L , Lemma 2, and the definition of \models , we have: $g \models^{I_0} L(r, c)$ iff $g \models^{I_0} L'(r, c, \emptyset)$ iff $h \models^{I_0} c$. \square

Transformation App , adapted from Def in [14], takes as input a rule set \mathcal{R} and generates an M-constraint that is satisfied by graphs for which \mathcal{R} is applicable.

Theorem 4 (Applicability of a rule). There is a transformation App such that for every rule set \mathcal{R} and every graph G ,

$$G \models \text{App}(\mathcal{R}) \text{ if and only if } \exists H. G \Rightarrow_{\mathcal{R}} H.$$

Construction. If \mathcal{R} is empty, define $\text{App}(\mathcal{R}) = \text{false}$; otherwise, for $\mathcal{R} = \{r_1, \dots, r_n\}$, define:

$$\text{App}(\mathcal{R}) = \text{app}(r_1) \vee \dots \vee \text{app}(r_n).$$

For each rule $r = \langle r', \text{ac} \rangle$ with $r' = \langle L \hookleftarrow K \hookrightarrow R \rangle$, we define $\text{app}(r) = \exists(\emptyset \hookrightarrow L, \text{Dang}(r') \wedge \text{ac}_L \wedge L(r, \text{ac}_R))$. Here, $\text{Dang}(r') = \bigwedge_{a \in A} \neg \exists a$, where the index set A ranges over all injective graph morphisms $a: L \hookrightarrow L^\oplus$ (up to isomorphic codomains) such that the pair $\langle K \hookrightarrow L, a \rangle$ has no pushout complement; each L^\oplus a graph that can be obtained from L by adding either (1) a loop; (2) a single edge between distinct nodes; or (3) a single node and a non-looping edge incident to that node. \square

Proof. See the corresponding proofs in [14] and [17] for nested conditions and E-conditions respectively. (The difference is in the application conditions, i.e. M-conditions over L and R . Correctness follows from the definition of \models for M-conditions and Theorem 3.) \square

Finally, transformation Pre (adapted from [8]) combines the other transformations to construct a weakest liberal precondition relative to a rule and postcondition.

Theorem 5 (Postconditions to weakest liberal preconditions). There is a transformation Pre such that for every rule $r = \langle \langle L \hookleftarrow K \hookrightarrow R \rangle, \text{ac} \rangle$, every M-constraint c , and every direct derivation $G \Rightarrow_r H$,

$$G \models \text{Pre}(r, c) \text{ if and only if } H \models c.$$

Moreover, $\text{Pre}(r, c) \vee \neg \text{App}(\{r\})$ is the *weakest liberal precondition* relative to r and c .

Construction. Let $r = \langle \langle L \hookleftarrow K \hookrightarrow R \rangle, \text{ac} \rangle$ denote a rule and c denote an M-constraint. Then:

$$\text{Pre}(r, c) = \forall(\emptyset \hookrightarrow L, (\text{Dang}(r) \wedge \text{ac}_L \wedge L(r, \text{ac}_R)) \Rightarrow L(r, A(r, c))).$$

\square

Proof. As for nested conditions (see [14]), but adapted for the definition of \models for M-conditions and Theorem 3. That Pre and App can be used to construct the weakest liberal precondition is shown in [17]. \square

Example 5. Take **grow**, c , γ and $L(\text{grow}, A(\text{grow}, c))$ as considered in Example 4. Applying transformation Pre:

$$\begin{aligned} & \text{Pre}(\text{grow}, L(\text{grow}, A(\text{grow}, c))) \\ &= \forall(\bullet_1, \text{ac}_L \Rightarrow \exists \mathbf{v} \mathbf{X}, \mathbf{Y} [\bigwedge_{i \in \{1,2,4\}} \forall(\bullet_i \hookrightarrow Y_i, \exists(Y_i \mid \text{path}(\mathbf{v}, \mathbf{w})) \Rightarrow \exists(Y_i \mid \gamma)) \\ & \quad \wedge \forall(\bullet_1 \bullet_{\mathbf{v}}, \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \text{path}, (\mathbf{v}, 1)) \Rightarrow \exists(\bullet_1 \bullet_{\mathbf{v}} \mid \mathbf{v} \in \mathbf{X} \text{ and not } \mathbf{v} \in \mathbf{Y})) \\ & \quad \wedge \forall(\bullet_{1=\mathbf{v}}, \exists(\bullet_{1=\mathbf{v}} \mid \mathbf{v} \in \mathbf{X} \text{ and not } \mathbf{v} \in \mathbf{Y}))]) \end{aligned}$$

where the graphs Y_i are as given in Figure 1. This M-constraint is only satisfied by graphs that do not have any edges between distinct nodes, because of the assertion that every match (i.e. every node) must be in \mathbf{X} and not in \mathbf{Y} . Were an edge to exist – i.e. a path – then the M-constraint asserts that its target is in \mathbf{Y} ; a contradiction. \square

6 Proving Non-Local Specifications

In this section we show how to systematically prove a non-local correctness specification using a Hoare logic adapted from [18,17]. The key difference is the use of M-constraints as assertions, and our extension of Pre in constructing weakest liberal preconditions for rules. (We note that one could just as easily adapt the Dijkstra-style systems of [8,14].)

We will specify the behaviour of programs using (*Hoare*) *triples*, $\{c\} P \{d\}$, where P is a program, and c, d are *pre-* and *postconditions* expressed as M-constraints. We say that this specification holds in the sense of *partial correctness*, denoted by $\models \{c\} P \{d\}$, if for any graph G satisfying c , every graph H resulting from the execution of P on G satisfies d .

For systematically proving a specification, we present a *Hoare logic* in Figure 2, where c, d, e, inv range over M-constraints, P, Q over programs, r over rules, and \mathcal{R} over rule sets. If a triple $\{c\} P \{d\}$ can be instantiated from an axiom or deduced from an inference rule, then it is *provable* in the Hoare logic and we write $\vdash \{c\} P \{d\}$. Proofs shall be displayed as trees, with the specification as the root, axiom instances as the leaves, and inference rule instances in-between.

For simplicity in proofs we will typically treat $[\text{ruleapp}]_{\text{wlp}}$ as two different axioms (one for each disjunct). Note that we have omitted, due to space, the proof rules for the conditional constructs. Note also the restriction to rule sets in [!], because the applicability of arbitrary programs cannot be expressed in a logic for which the model checking problem is decidable [17].

Theorem 6 (Soundness). Given a program P and M-constraints c, d , we have that $\vdash \{c\} P \{d\}$ implies $\models \{c\} P \{d\}$. \square

$$\begin{array}{c}
[\text{ruleapp}]_{\text{wlp}} \quad \frac{\{\text{Pre}(r, c) \vee \neg \text{App}(\{r\})\} \quad r \quad \{c\}}{\{c\} \quad r \quad \{d\} \text{ for each } r \in \mathcal{R}} \quad [\text{ruleset}] \quad \frac{\{c\} \quad r \quad \{d\} \text{ for each } r \in \mathcal{R}}{\{c\} \quad \mathcal{R} \quad \{d\}} \\
[\text{comp}] \quad \frac{\{c\} \quad P \quad \{e\} \quad \{e\} \quad Q \quad \{d\}}{\{c\} \quad P; Q \quad \{d\}} \quad [!] \quad \frac{\{inv\} \quad \mathcal{R} \quad \{inv\}}{\{inv\} \quad \mathcal{R}! \quad \{inv \wedge \neg \text{App}(\mathcal{R})\}} \\
[\text{cons}] \quad \frac{c \Rightarrow c' \quad \{c'\} \quad P \quad \{d'\} \quad d' \Rightarrow d}{\{c\} \quad P \quad \{d\}}
\end{array}$$

Fig. 2. A Hoare logic for partial correctness

Proof. See [17] for a soundness proof of the corresponding extensional partial correctness calculus. \square

The remainder of this section demonstrates the use of our constructions and Hoare logic in proving non-local specifications of two programs. For the first, we will consider a property expressed in terms of MSO variables and expressions, whereas for the second, we will consider properties expressed in terms of **path** predicates. Both programs are simple, as our focus here is not on building intricate proofs but rather on illustrating the main novelty of this paper: a Pre construction for MSO properties.

Example 6. Recall the program **init**; **grow**! of Example 2 that nondeterministically constructs a tree. A known non-local property of trees is that they can be assigned a 2-colouring (i.e. they are bipartite), a property that the M-constraint *col* of Example 1 precisely expresses. Hence we will show that $\vdash \{emp\} \text{init}; \text{grow}! \{col\}$, where $emp = \neg \exists (\bullet)$ expresses that the graph is empty. A proof tree for this specification is given in Figure 3, where the interpretation constraints γ_1 and γ_2 in $\text{Pre}(\text{grow}, col)$ are respectively $(v \in X \text{ or } v \in Y)$ and $\text{not } (v \in X \text{ and } v \in Y)$ and $\text{not } (v \in X \text{ and } w \in X)$ and $\text{not } (v \in Y \text{ and } w \in Y)$.

Observe that $\text{Pre}(\text{grow}, col)$ is essentially an “embedding” of the postcondition *col* within the context of possible matches for **grow**. The second line expresses that every node (whether the node of the match or not) is coloured **X** or **Y**. The following three conjuncts then express that any edges in the various contexts of the match connect nodes that are differently coloured. The final conjunct is of the same form, but is “pre-empting” the creation of a node and edge by **grow**. To ensure that the graph remains 2-colourable, node 1 of the match must not belong to both sets; this, of course, is already established by the first nested conjunct. Hence the first implication arising from instances of $[\text{cons}]$, $col \Rightarrow \text{Pre}(\text{grow}, col)$, is valid. The second implication, $emp \Rightarrow \text{Pre}(\text{init}, col)$, is also valid since a graph satisfying *emp* will not have any nodes to quantify over. \square

Example 7. An *acyclic graph* is a graph that does not contain any *cycles*, i.e. non-empty paths starting and ending on the same node. One way to test for acyclicity is to apply the rule **delete** = $\langle \langle \bullet_1 \rightarrow \bullet_2 \Rightarrow \bullet_1 \bullet_2 \rangle, ac_L \rangle$ for as long as

$$\frac{\frac{\{Pre(init, col)\} \text{ init } \{col\}}{\{emp\} \text{ init } \{col\}} \quad \frac{\frac{\{Pre(grow, col)\} \text{ grow } \{col\}}{\{col\} \text{ grow } \{col\}}}{\{col\} \text{ grow! } \{col \wedge \neg App(\{grow\})\}} \quad \vdash \{emp\} \text{ init; grow! } \{col\}$$

$$Pre(init, col) \equiv col$$

$$Pre(grow, col) \equiv \forall(\bullet_1, \neg tc \Rightarrow \exists_v X, Y[\\ \forall(\bullet_1 \bullet_v, \exists(\bullet_1 \bullet_v \mid \gamma_1)) \wedge \forall(\bullet_{1=v}, \exists(\bullet_{1=v} \mid \gamma_1)) \\ \wedge \forall(\bullet_1 \bullet_v \bullet_w, \exists(\bullet_1 \bullet_v \rightarrow \bullet_w) \Rightarrow \exists(\bullet_1 \bullet_v \bullet_w \mid \gamma_2)) \\ \wedge \forall(\bullet_{1=v} \bullet_w, \exists(\bullet_{1=v} \rightarrow \bullet_w) \Rightarrow \exists(\bullet_{1=v} \bullet_w \mid \gamma_2)) \\ \wedge \forall(\bullet_{1=w} \bullet_v, \exists(\bullet_{1=w} \leftarrow \bullet_v) \Rightarrow \exists(\bullet_{1=w} \bullet_v \mid \gamma_2)) \\ \wedge (\forall(\bullet_{1=v}, \exists(\bullet_{1=v} \mid \text{not } v \in X)) \\ \vee \forall(\bullet_{1=v}, \exists(\bullet_{1=v} \mid \text{not } v \in Y)))])$$

Fig. 3. Trees are 2-colourable

possible; the resulting graph being edgeless if the input graph was acyclic. Here, ac_L denotes the left application condition $\neg \exists(\bullet_1 \rightarrow \bullet_2 \hookrightarrow \bullet_1 \rightarrow \bullet_2) \vee \neg \exists(\bullet_1 \rightarrow \bullet_2 \hookrightarrow \bullet_1 \rightarrow \bullet_2)$, expressing that in matches, either the source node has indegree 0 or the target node has outdegree 0 (we do not consider the special case of looping edges for simplicity). Note that nodes *within* a cycle would not satisfy this: if a source node has an indegree of 0 for example, there would be no possibility of an outgoing path ever returning to the same node.

We prove two claims about this rule under iteration: first, that it deletes all edges in an acyclic graph; second, that if applied to a graph containing cycles, the resulting graph would not be edgeless. That is, $\vdash \{-c\} \text{ delete! } \{e\}$ and $\vdash \{c\} \text{ delete! } \{-e\}$, for M-constraints c (for cycles), e (for edgeless), $\gamma_c = \text{path}(v, w, \text{not } e)$ and $\text{path}(w, v, \text{not } e)$, and proofs as in Figure 4.

First, observe that $Pre(\text{delete}, \neg c)$ is essentially an “embedding” of the postcondition $\neg c$ within the context of possible matches for delete . The path predicates in γ_c now additionally assert (as a result of the L transformation) that paths do not include images of edge e : this is crucially important for establishing the postcondition because the rule deletes the edge. For space reasons we did not specify $Pre(\text{delete}, c)$, but this can be constructed from $Pre(\text{delete}, \neg c)$ by replacing each \wedge with \vee and removing each \neg in the nested part.

The instances of [cons] give rise to implications that we must show to be valid. First, $\neg c \Rightarrow Pre(\text{delete}, \neg c)$ is valid: a graph satisfying $\neg c$ does not contain any cycles, hence it also does not contain cycles outside of the context of matches for delete . Second, $\neg c \wedge \neg App(\{\text{delete}\}) \Rightarrow e$ is valid: a graph satisfying the antecedent does not contain any cycles and also no pair of incident nodes for which ac_L holds. If the graph is not edgeless, then there must be some such pair satisfying ac_L ; otherwise the edges are within a cycle. Hence the graph must be edgeless, satisfying e .

$$\begin{array}{c}
\frac{\frac{\frac{\{\text{Pre}(\text{delete}, \neg c)\} \text{ delete } \{\neg c\}}{\{\neg c\} \text{ delete } \{\neg c\}}}{\{\neg c\} \text{ delete! } \{\neg c \wedge \neg \text{App}(\{\text{delete}\})\}}} \\
\vdash \{\neg c\} \text{ delete! } \{e\}
\end{array}
\qquad
\begin{array}{c}
\frac{\frac{\frac{\{\text{Pre}(\text{delete}, c)\} \text{ delete } \{c\}}{\{c\} \text{ delete } \{c\}}}{\{c\} \text{ delete! } \{c \wedge \neg \text{App}(\{\text{delete}\})\}}} \\
\vdash \{c\} \text{ delete! } \{\neg e\}
\end{array}$$

$$\begin{aligned}
c &= \exists (\bullet_v \bullet_w \mid \text{path}(v, w) \text{ and } \text{path}(w, v)) \\
e &= \neg \exists (\bullet_v \xrightarrow{e} \bullet_w) \\
\text{Pre}(\text{delete}, \neg c) &= \forall (\bullet_1 \xrightarrow{e} \bullet_2, \text{ac}_L \Rightarrow \\
&\quad \wedge \neg \exists (\bullet_1 \xrightarrow{e} \bullet_2 \bullet_v \bullet_w \mid \gamma_c) \wedge \neg \exists (\bullet_{1=v} \xrightarrow{e} \bullet_2 \bullet_w \mid \gamma_c) \\
&\quad \wedge \neg \exists (\bullet_1 \xrightarrow{e} \bullet_{2=v} \bullet_w \mid \gamma_c) \wedge \neg \exists (\bullet_{1=w} \xrightarrow{e} \bullet_2 \bullet_v \mid \gamma_c) \\
&\quad \wedge \neg \exists (\bullet_1 \xrightarrow{e} \bullet_{2=w} \bullet_v \mid \gamma_c) \wedge \neg \exists (\bullet_{1=v} \xrightarrow{e} \bullet_{2=w} \mid \gamma_c) \\
&\quad \wedge \neg \exists (\bullet_{1=w} \xrightarrow{e} \bullet_{2=v} \mid \gamma_c)) \\
\text{App}(\{\text{delete}\}) &= \exists (\bullet_1 \xrightarrow{e} \bullet_2, \text{ac}_L)
\end{aligned}$$

Fig. 4. Acyclicity (or lack thereof) is invariant

In the second proof tree, $c \Rightarrow \text{Pre}(\text{delete}, c)$ is valid. A graph satisfying c contains a cycle: clearly, no edge (with its source and target) in this cycle satisfies ac_L ; hence the graph satisfies the consequent, since images of edge e cannot be part of the cycle in the graph. Finally, $c \wedge \neg \text{App}(\{\text{delete}\}) \Rightarrow \neg e$ is valid: if a graph satisfies the antecedent, then it contains a cycle, the edges of which **delete** will never be applicable to because of ac_L ; hence the graph cannot be edgeless, and satisfies $\neg e$. \square

7 Related Work

We point to a few related publications addressing the verification of non-local graph properties through proofs / theorem proving and model checking.

Habel and Radke have considered HR conditions [9], an extension of nested conditions embedding hyperedge replacement grammars via graph variables. The formalism is more expressive than MSO logic on graphs (it is able, for example, to express node-counting MSO properties such as “the graph has an even number of nodes” [21]) but it is not yet clear whether an effective construction for weakest liberal preconditions exists. Percebois et al. [15] demonstrate how one can verify global invariants involving paths, directly at the level of rules. Rules are modelled with (a fragment of) first-order logic on graphs in the interactive theorem prover Isabelle. Inaba et al. [10] address the verification of type-annotated Core UnCAL – a query algebra for graph-structured databases – against input/output graph schemas in MSO. They first reformulate the query algebra itself in MSO, before applying an algorithm that reduces the verification problem to the validity of MSO over trees.

The GROOVE model checker [6] supports rules with paths in the left-hand side, expressed as a regular expression over edge labels. One can specify such rules to match only when some (un)desirable non-local property holds, and then

verify automatically that the rule is never applicable. Augur 2 [11] also uses regular expressions, but for expressing forbidden paths that should not occur in any reachable graph.

8 Conclusion

This paper has contributed the means for systematic proofs of graph programs with respect to non-local specifications. In particular, we defined M-conditions, an extension of nested conditions equivalently expressive to MSO logic on graphs, and defined for this assertion language an effective construction for weakest liberal preconditions of rules. We demonstrated the use of this work in some Hoare-style proofs of programs relative to non-local invariants, i.e. the existence of 2-colourings, and the existence of arbitrary-length cycles. Some interesting topics for future work include: extending M-conditions and Pre to support other useful predicates (e.g. an *undirected* path predicate), adding support for attribution (e.g. along the lines of [18,17]), implementing the construction of Pre, and generalising the resolution- and tableau-based reasoning systems for nested conditions [13,12] to M-conditions.

Acknowledgements. The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

References

1. Courcelle, B.: Graph rewriting: An algebraic and logic approach. In: Handbook of Theoretical Computer Science, vol. B, chap. 5. Elsevier (1990)
2. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation* 85(1), 12–75 (1990)
3. Courcelle, B., Engelfriet, J.: *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press (2012)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation* (Monographs in Theoretical Computer Science. An EATCS Series). Springer (2006)
5. Flum, J., Grohe, M.: *Parameterized Complexity Theory*. Springer (2006)
6. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer* 14(1), 15–40 (2012)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2), 245–296 (2009)
8. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: *Proc. Graph Transformations (ICGT 2006)*. LNCS, vol. 4178, pp. 445–460. Springer (2006)
9. Habel, A., Radke, H.: Expressiveness of graph conditions with variables. In: *Proc. International Colloquium on Graph and Model Transformation (GraMoT 2010)*. Electronic Communications of the EASST, vol. 30 (2010)

10. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: Proc. Principles and Practice of Declarative Programming (PPDP 2011). pp. 17–28. ACM (2011)
11. König, B., Kozioura, V.: Augur 2 - a new version of a tool for the analysis of graph transformation systems. In: Proc. International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). ENTCS, vol. 211, pp. 201–210. Elsevier (2008)
12. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Proc. International Conference on Graph Transformation (ICGT 2014). LNCS, vol. 8571. Springer (2014), to appear.
13. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In: Proc. International Conference on Graph Transformations (ICGT 2008). LNCS, vol. 5214, pp. 289–304. Springer (2008)
14. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Universität Oldenburg (2009)
15. Percebois, C., Strecker, M., Tran, H.N.: Rule-level verification of graph transformations for invariants based on edges' transitive closure. In: Proc. Software Engineering and Formal Methods (SEFM 2013). LNCS, vol. 8137, pp. 106–121. Springer (2013)
16. Plump, D.: The design of GP 2. In: Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). Electronic Proceedings in Theoretical Computer Science, vol. 82, pp. 1–16 (2012)
17. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, University of York (2013)
18. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. *Fundamenta Informaticae* 118(1-2), 135–175 (2012)
19. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. In: Revised Selected Papers, Graph Computation Models (GCM 2012). Electronic Communications of the EASST, vol. 61 (2013)
20. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Proc. International Conference on Graph Transformation (ICGT 2014). LNCS, vol. 8571. Springer (2014), to appear.
21. Radke, H.: HR^* graph conditions between counting monadic second-order and second-order graph formulas. In: Revised Selected Papers, Graph Computation Models (GCM 2012). Electronic Communications of the EASST, vol. 61 (2013)

Appendix: Proofs and Semantics

A Expressive Equivalence to MSO Formulae

In this section we prove that M-conditions and MSO formulae on graphs are equivalently expressive. We define a many-sorted MSO logic on graphs (in the spirit of [1]), and show that there are translations from this logic to M-conditions and vice versa. The logic and translations are based on those of [17] for nested conditions with expressions. (An alternative approach is to use a single-sorted logic, e.g. [9].) Throughout this section we will assume that graphs are labelled over some fixed label alphabet $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$.

A.1 Syntax and Semantics

We define the syntax and semantics of a many-sorted MSO logic on graphs. The idea is to assign sorts (or types) – edge, vertex, edge set, or vertex set – to every expression of the logic, and prevent at the syntactic level the composition of formulae that do not “make sense” under interpretation. For example, we discard as syntactically ill-formed any expression $\mathbf{s}(x)$ in which x is not an edge expression (since this will be interpreted as the source function of some graph).

Definition 7 (Expressions). The grammar in Figure 5 defines four syntactic categories of *expressions*: Edge, Vertex, EdgeSet, and VertexSet. They respectively contain (disjoint) syntactic categories of *variables*: EVar, VVar, ESetVar, and VSetVar.

```

Expression ::= Edge | Vertex | EdgeSet | VertexSet
Edge       ::= EVar
Vertex     ::= VVar | (s | t) ' (' Edge ')'
EdgeSet    ::= ESetVar
VertexSet  ::= VSetVar

```

Fig. 5. Abstract syntax of expressions

□

Definition 8 (Sorts, sort function). Every expression is associated with a *sort* (or *type*), determined by the syntactic category it is contained within. We use the name of that category to denote its sort. The function $\text{sort}(e)$ is the *sort function*, that takes an expression e as input and returns its sort. □

The formulae of the logic can quantify over first-order and MSO (i.e. set) variables, and express the existence of edges and nodes in sets of the corresponding type. Note that we do not include equality of set variables, since this can be defined precisely in terms of set membership over individual elements.

```

Formula ::= true | false | Edge '=' Edge | Vertex '=' Vertex
         | labb '(' Edge ')' | labc '(' Vertex ')'
         | Edge '∈' EdgeSet | Vertex '∈' VertexSet
         | '¬' Formula | Formula BoolOp Formula
         | Quantifier (VVar ':' V' | EVar ':' E'
                       | ESetVar ':' ES' | VSetVar ':' VS' ) '.' Formula
BoolOp   ::= ∧ | ∨ | ⇒ | ⇔
Quantifier ::= ∀ | ∃

```

Fig. 6. Abstract syntax of formulae

Definition 9 (Formulae). Figure 6 defines *formulae*, where $b \in \mathcal{C}_E$ and $c \in \mathcal{C}_V$. \square

The symbols \mathbf{s}, \mathbf{t} are *function symbols* of arity one, and are syntactic representations of source and target functions. The symbols lab_y are *predicate symbols* of arity one, expressing that an item is labelled by y . The symbols $=, \in$ are predicate symbols of arity two, and are syntactic representations of equality and set membership.

The *free variables* of a formula are those that are not bound by a quantifier. Note that such variables still have sorts. If a formula contains no such free variables, then we call it a sentence.

Definition 10 (Sentence). A *sentence* (or a *closed formula*) is a formula that contains no free variables. \square

Sentences of the logic are evaluated with respect to interpretations. These map the sorts to disjoint semantic domains, function symbols to functions, and predicate symbols to Boolean-valued functions. In particular, given some graph, we build an interpretation from its nodes, edges, source, target, and labelling functions. (Note that interpretations here are different from interpretations for M-conditions, which map only set variables to elements of the corresponding semantic domains.)

Definition 11 (Satisfaction of sentences). An *interpretation* I is a mapping from (1) sorts to semantic domains, (2) expressions $f(e_1, \dots, e_n)$, with f a function symbol and each e_i an expression, to functions of arity:

$$I(\text{sort}(e_1)) \times \dots \times I(\text{sort}(e_n)) \rightarrow I(\text{sort}(f(e_1, \dots, e_n))),$$

and (3) formulae $p(e_1, \dots, e_n)$, with p a predicate symbol and each e_i an expression, to Boolean-valued functions of arity:

$$I(\text{sort}(e_1)) \times \dots \times I(\text{sort}(e_n)) \rightarrow \mathbb{B}.$$

Let I be an interpretation function, and φ be a sentence. The *satisfaction* of φ by I , denoted $I \models \varphi$, is defined inductively as follows.

If φ is **true** (resp. **false**), then $I \models \varphi$ (resp. $I \models \varphi$ does not hold). If φ is $p(e_1, \dots, e_n)$ with p a predicate symbol and each e_i an expression, then $I \models \varphi$ if $I(p)(I(e_1), \dots, I(e_n)) = \text{true}$.

Let φ_1, φ_2 be sentences. If φ is $\neg\varphi_1$, then $I \models \varphi$ if $I \models \varphi_1$ does not hold. If φ is $\varphi_1 \wedge \varphi_2$ (resp. $\varphi_1 \vee \varphi_2$), then $I \models \varphi$ if $I \models \varphi_1$ and (resp. or) $I \models \varphi_2$. If φ is $\varphi_1 \Rightarrow \varphi_2$, then $I \models \varphi$ if $I \models \neg\varphi_1$ or $I \models \varphi_2$. If φ is $\varphi_1 \Leftrightarrow \varphi_2$, then $I \models \varphi$ if $I \models \varphi_1 \Rightarrow \varphi_2$ and $I \models \varphi_2 \Rightarrow \varphi_1$.

Let \mathbf{x} be a variable of sort s , and φ_1 be a formula with \mathbf{x} as its only free variable. Let also S denote the symbol that corresponds with sort s . If φ has the form $\exists \mathbf{x} : S. \varphi_1$, then $I \models \varphi$ if there is some $a \in I(s)$ such that $I_{\mathbf{x} \mapsto a} \models \varphi_1$ where $I_{\mathbf{x} \mapsto a}$ is equal to I but with the addition that $I(\mathbf{x}) = a$. If φ is $\forall \mathbf{x} : S. \varphi_1$, then $I \models \varphi$ if for every $a \in I(s)$, $I_{\mathbf{x} \mapsto a} \models \varphi_1$. □

Definition 12 (Satisfaction of sentences by graphs). Let G be a graph and φ be a sentence. We say that G *satisfies* φ , denoted by $G \models \varphi$, if $I_G \models \varphi$, where I_G is the *interpretation induced by* G , defined as follows:

Sorts. We define $I_G(\text{Edge}) = E_G$, $I_G(\text{Vertex}) = V_G$, $I_G(\text{EdgeSet}) = 2^{E_G}$, and $I_G(\text{VertexSet}) = 2^{V_G}$.

Function symbols. We define $I_G(\mathbf{s}) = s_G$ and $I_G(\mathbf{t}) = t_G$. We define $I_G(1)$ and $I_G(\mathbf{m})$ to be the functions l_G and m_G respectively.

Predicate symbols. We define $I_G(\text{lab}_b) = \text{lab}_b$ where $\text{lab}_b : E_G \rightarrow \mathbb{B}$ returns true for inputs e if $m_G(e) = b$; false otherwise. (Analogous for node label predicates.) We define $I_G(=)$ to be equality in the standard sense. We define $I_G(\in) = \text{in}_G$ where $\text{in}_G : (E_G \times 2^{E_G}) \cup (V_G \times 2^{V_G}) \rightarrow \mathbb{B}$ returns true for inputs (x, X) if $x \in X$; false otherwise. □

A.2 From Formulae to M-Conditions

In this subsection we prove that formulae can be translated into equivalent M-conditions. We define a translation over the abstract syntax of formulae and expressions. It is assumed that distinct quantifiers bind distinct variables in formulae, allowing us to use node and edge variables as identifiers in the corresponding M-condition. This correspondence is very important in the translation: a node variable \mathbf{v} will correspond to a node identifier v in the M-condition, and an edge variable \mathbf{e} will correspond to an edge identifier e with source and target nodes s_e, t_e .

First, we define a helper function that takes a Vertex-sorted expression as input, and returns the node identifier that will be associated with it in the M-condition.

Definition 13 (Helper function VertexID). Let t denote an expression in Vertex. We define:

$$\text{VertexID}(t) = \begin{cases} v & \text{if } t = \mathbf{v} \text{ with } \mathbf{v} \in \text{VVar} \\ s_e & \text{if } t = \mathbf{s}(\mathbf{e}) \text{ with } \mathbf{e} \in \text{EVar} \\ t_e & \text{if } t = \mathbf{t}(\mathbf{e}) \text{ with } \mathbf{e} \in \text{EVar} \end{cases}$$

□

Theorem 7 (Sentences can be expressed as M-constraints). Let φ denote a sentence. There is a transformation Cond such that for all graphs G ,

$$G \models \varphi \text{ if and only if } G \models \text{Cond}(\varphi).$$

Construction. We assume that quantifiers in φ bind distinct variables (otherwise one can always rename the variables), which allows for variables to correspond to node and edge identifiers. For all sentences φ , let $\text{Cond}(\varphi) = \text{Cond}'(\varphi, \emptyset)$. The transformation Cond' takes the formula that remains to be translated as its first input, and the domain of the next morphism in the generated M-condition as its second input. We define it inductively over the abstract syntax of formulae (Figure 6) and expressions (Figure 5).

Let X denote a graph over \mathcal{C} . Let $\varphi', \varphi_1, \varphi_2$ denote formulae (not necessarily sentences).

If $\varphi = \mathbf{true}$ (resp. \mathbf{false}), then $\text{Cond}'(\varphi, X) = \mathbf{true}$ (resp. \mathbf{false}). If $\varphi = \neg\varphi'$, then $\text{Cond}'(\varphi, X) = \neg\text{Cond}'(\varphi', X)$. If $\varphi = \varphi_1 \oplus \varphi_2$ with $\oplus \in \text{BoolOp}$, then $\text{Cond}'(\varphi, X) = \text{Cond}'(\varphi_1, X) \oplus \text{Cond}'(\varphi_2, X)$.

If $\varphi = e = f$ with $e, f \in \text{EVar}$, then $\text{Cond}'(\varphi, X) = \mathbf{true}$ if edges e, f are identified in X , otherwise \mathbf{false} .

If $\varphi = v_1 = v_2$ with v_1, v_2 in Vertex , then $\text{Cond}'(\varphi, X) = \mathbf{true}$ if $\text{VertexID}(v_1), \text{VertexID}(v_2)$ are identified in X , otherwise \mathbf{false} .

If $\varphi = \text{lab}_b(e)$ with $b \in \mathcal{C}_E$ and e in EVar , then $\text{Cond}'(\varphi, X) = \mathbf{true}$ if $m_X(e) = b$, otherwise \mathbf{false} . (Analogous for node label predicates.)

If $\varphi = e \in E$ with e in EVar and E in ESetVar , then:

$$\text{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid e \in E).$$

If $\varphi = v \in V$ with v in Vertex and V in VSetVar , then:

$$\text{Cond}'(\varphi, X) = \exists(X \hookrightarrow X \mid \text{VertexID}(v) \in V).$$

If $\varphi = \exists v : V. \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \bigvee_{X' \in \text{VMerge}(X, v)} \exists(X \hookrightarrow X', \text{Cond}'(\varphi', X'))$$

Here, $\text{VMerge}(X, v)$ is the (finite) set of graphs constructed from X by disjointly adding a single node v with some label in \mathcal{C}_V , and every graph obtainable from these by identifying a node with v .

If $\varphi = \exists e : E. \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \bigvee_{X' \in \text{EMerge}(X, e)} \exists(X \hookrightarrow X', \text{Cond}'(\varphi', X'))$$

Here, $\text{EMerge}(X, e)$ is the (finite) set of graphs defined as follows. Let X^* denote a graph obtained from X by disjointly adding nodes with identifiers s_e, t_e and

an edge with identifier e such that $s_{X^*}(e) = s_e$, $t_{X^*}(e) = t_e$, $l_{X^*}(s_e) \in \mathcal{C}_V$, $l_{X^*}(t_e) \in \mathcal{C}_V$, and $m_{X^*}(e) \in \mathcal{C}_E$. The set $\text{EMerge}(X, \mathbf{e})$ contains all such graphs X^* , and all other graphs obtainable from them by identifying e, s_e, t_e with nodes and edges. (Note that s_e and t_e can be identified to create a loop.)

If $\varphi = \exists \mathbf{E} : \text{ES} . \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \exists_{\mathbf{E}} \mathbf{E} [\text{Cond}'(\varphi', X)]$$

If $\varphi = \exists \mathbf{V} : \text{VS} . \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \exists_{\mathbf{V}} \mathbf{V} [\text{Cond}'(\varphi', X)]$$

If $\varphi = \forall \mathbf{x} : S . \varphi'$, then:

$$\text{Cond}'(\varphi, X) = \text{Cond}'(\neg \exists \mathbf{x} : S . \neg \varphi', X).$$

□

To prove the theorem, we first prove a more general lemma about the translation of formulae.

Lemma 3 (Formulae can be expressed as M-conditions). Let φ denote a formula, I an interpretation defined for the free set variables of φ , and X a graph in which every identifier x corresponds to a free (node or edge) variable \mathbf{x} in φ . For all injective graph morphisms $z : X \hookrightarrow G$, we have:

$$I_G^{z,I} \models \varphi \text{ if and only if } z : X \hookrightarrow G \models^I \text{Cond}'(\varphi, X).$$

Here, $I_G^{z,I}$ is defined as I_G but with the following mappings for free variables in φ : (1) for each set variable \mathbf{Y} in the domain of I , $I_G^{z,I}(\mathbf{Y}) = I(\mathbf{Y})$; (2) for each node v in X , $I_G^{z,I}(\mathbf{v}) = z(v)$; and (3) for each edge e in X , $I_G^{z,I}(\mathbf{e}) = z(e)$. □

Proof. Basis. Most cases are easily adapted from the proof of Lemma 6.18 in [17]. In the case that φ has the form $\text{lab}_b(\mathbf{e})$ with \mathbf{e} in EVar ,

$$\begin{aligned} (I_G^{z,I} \models \text{lab}_b(\mathbf{e})) &= I_G^{z,I}(\text{lab}_b)(\mathbf{e}) \\ &= (\text{lab}_b(z(e))) \\ &= (m_G(z(e)) = b) \\ &= (m_X(e) = b) \\ &= (z \models^I \text{Cond}'(\varphi, X)) \end{aligned}$$

(Analogous for node label predicates.)

In the case that φ has the form $\mathbf{e} \in \mathbf{E}$ with \mathbf{e} in EVar and \mathbf{E} in ESetVar ,

$$\begin{aligned} (I_G^{z,I} \models \mathbf{e} \in \mathbf{E}) &= I_G^{z,I}(\in)(\mathbf{e}, \mathbf{E}) \\ &= (\text{in}_G(z(e), I(\mathbf{E}))) \\ &= (z(e) \in I(\mathbf{E})) \\ &= (e \in \mathbf{E})^{I,z} \\ &= (z \models^I \exists (X \hookrightarrow X \mid e \in \mathbf{E})) \\ &= (z \models^I \text{Cond}'(\varphi, X)) \end{aligned}$$

Step. Only if. Assume that $I_G^{z,I} \models \varphi$. Most cases are easily adapted from the proof of Lemma 6.18 in [17]. In the case that φ has the form $\exists E : ES. \varphi'$, by assumption, there exists some $a \in 2^{E_G}$ such that $I_G^{z,I} \cup \{E \mapsto a\} \models \varphi'$. Define $I' = I \cup \{E \mapsto a\}$. Then $I_G^{z,I'} \models \varphi'$, and by induction hypothesis, $z \models^{I'} \text{Cond}'(\varphi', X)$. Finally, with the definition of \models for M-conditions, we get the result that $z \models^I \exists_E E [\text{Cond}'(\varphi', X)] = \text{Cond}'(\varphi, X)$. (Case for node set quantification is analogous.)

Step. If. Assume that $z \models^I \text{Cond}'(\varphi, X)$. Most cases are easily adapted from the proof of Lemma 6.18 in [17]. In the case that φ has the form $\exists E : ES. \varphi'$, by assumption and construction, we have $z \models^I \exists_E E [\text{Cond}'(\varphi', X)]$. Then there is some $I' = I \cup \{E \mapsto a\}$ with $a \subseteq E_G$ (equiv. $a \in 2^{E_G}$) such that $z \models^{I'} \text{Cond}'(\varphi', X)$. By induction hypothesis, we have $I_G^{z,I'} \models \varphi'$. Finally, with the definition of \models for formulae, we have the result that $I_G^{z,I} \models \exists E : ES. \varphi'$. (Case for node set quantification is analogous.) \square

Proof (of Theorem 7). Define $i_G : \emptyset \hookrightarrow G$. We have that:

$$\begin{aligned} G \models \varphi & \text{ iff } I_G \models \varphi \\ & \text{ iff } I_G^{i_G, I_\emptyset} \models \varphi \\ & \text{ iff } i_G \models^{I_\emptyset} \text{Cond}'(\varphi, \emptyset) \\ & \text{ iff } i_G \models^{I_\emptyset} \text{Cond}(\varphi) \\ & \text{ iff } G \models \text{Cond}(\varphi). \end{aligned}$$

from the definition of \models for formulae and M-conditions, and Lemma 3. \square

A.3 From M-Conditions to Formulae

In this subsection we prove that M-conditions can be translated into equivalent formulae. To simplify the translation, we define a normal form for M-conditions that allows us to assume one new node or edge per level of nesting, as well as the absence of path predicates.

Definition 14 (Normal form for M-conditions). An M-condition c is in *normal form* if all of the following hold:

1. all morphisms are inclusions;
2. all morphisms $a : P \hookrightarrow C$ are either identity morphisms ($P = C$), or C is the graph P but with one additional node or one additional edge;
3. no interpretation constraint contains a path predicate.

\square

Proposition 2 (M-conditions can be normalised). For every M-condition c , there is an M-condition \bar{c} in normal form such that c and \bar{c} are equivalent, i.e. for every morphism p , and every interpretation I ,

$$p \models^I c \text{ if and only if } p \models^I \bar{c}.$$

\square

Proof (sketch). Section 6.4.1 of [17] shows how morphisms can be replaced and decomposed to satisfy (1) and (2) of normal form. For (3), observe that an M-condition $\exists(a : X \hookrightarrow X \mid \text{path}(\mathbf{v}, \mathbf{w}, \text{not } e_1 \mid e_2 \mid \dots))$ can be replaced by the equivalent M-condition $\text{Cond}'(\varphi, X)$, where φ is defined as follows:

$$\begin{aligned} \varphi = \forall \mathbf{X} : \mathbf{VS}. \quad & ((\forall \mathbf{y}, \mathbf{z} : \mathbf{V}. \mathbf{y} \in \mathbf{X} \wedge (\exists \mathbf{e} : \mathbf{E}. \mathbf{s}(\mathbf{e}) = \mathbf{y} \wedge \mathbf{t}(\mathbf{e}) = \mathbf{z}) \\ & \wedge \neg \mathbf{e} = \mathbf{e}_1 \wedge \neg \mathbf{e} = \mathbf{e}_2 \wedge \dots) \Rightarrow \mathbf{z} \in \mathbf{X}) \\ & \wedge (\forall \mathbf{y} : \mathbf{V}. (\exists \mathbf{e} : \mathbf{E}. \mathbf{s}(\mathbf{e}) = \mathbf{v} \wedge \mathbf{t}(\mathbf{e}) = \mathbf{y}) \\ & \wedge \neg \mathbf{e} = \mathbf{e}_1 \wedge \neg \mathbf{e} = \mathbf{e}_2 \wedge \dots) \Rightarrow \mathbf{y} \in \mathbf{X})) \\ & \Rightarrow \mathbf{w} \in \mathbf{X}) \end{aligned}$$

i.e. path predicates can be expressed in terms of MSO expressions. \square

Now, we define and prove the correctness of a translation from M-conditions (in normal form) to formulae. The assumption that morphisms are inclusions allows us to establish a correspondence between identifiers and variables. For example, a node with identifier v will be translated into a variable \mathbf{v} from \mathbf{VVar} .

Theorem 8 (M-conditions can be expressed as formulae). There is a transformation Form such that for all M-constraints c , and all graphs G , we have:

$$G \models c \text{ if and only if } G \models \text{Form}(c).$$

Construction. We assume that M-constraint c is in normal form (otherwise replace it with an equivalent M-constraint that is). Define $\text{Form}(c) = \text{Form}'(c, \{\})$. Here, the second parameter can be understood as the set of all node and edge variables that have already been bound to quantifiers by the transformation. Then $\text{Form}'(c, V)$ is defined inductively as follows, where V denotes a set of sorted variables.

If $c = \mathbf{true}$, then $\text{Form}'(c, V) = \mathbf{true}$. If $c = \exists \mathbf{v} \mathbf{X}[c']$, then $\text{Form}'(c, V) = \exists \mathbf{X} : \mathbf{VS}. \text{Form}'(c', V)$ (analogous for edge set quantification). If $c = \exists(a \mid \gamma, c')$, then there are three possible outputs for $\text{Form}'(c, V)$ defined for the three forms that a may take in normal form.

Suppose that $c = \exists(\text{id}_P : P \hookrightarrow P \mid \gamma, c')$, i.e. an M-condition with a morphism that is an identity. Then, $\text{Form}'(c, V)$ is equal to:

$$\gamma^* \wedge \text{Form}'(c', V).$$

Here (and in the following), γ^* denotes the formula obtained from γ by replacing node identifiers v (resp. edge identifiers e) with variables \mathbf{v} in \mathbf{VVar} (resp. \mathbf{e} in \mathbf{EVar}), and by replacing **and**, **or**, **not** respectively with \wedge , \vee , \neg .

Suppose that $c = \exists([va] : P \hookrightarrow P' \mid \gamma, c')$, where $[va]$ denotes a morphism with codomain P' equal to domain P , except for an additional node v labelled with $a \in \mathcal{C}_V$. Then, $\text{Form}'(c, V)$ is equal to:

$$\exists \mathbf{v} : \mathbf{V}. (\bigwedge_{\mathbf{v}' \in V \cap \text{VVar}} \neg \mathbf{v} = \mathbf{v}') \wedge \text{lab}_a(\mathbf{v}) \wedge \gamma^* \wedge \text{Form}'(c', V \cup \{\mathbf{v}\})$$

Suppose that $c = \exists([euva]: P \hookrightarrow P' \mid \gamma, c')$, where $[euva]$ denotes a morphism with codomain P' equal to domain P , except for an additional edge e with label $a \in \mathcal{C}_E$, source node u , and target node v . Then, $\text{Form}'(c, V)$ is equal to:

$$\begin{aligned} \exists \mathbf{e} : \mathbf{E}. (\bigwedge_{\mathbf{e}' \in V \cap \text{EVar}} \neg \mathbf{e} = \mathbf{e}') \wedge \text{lab}_a(\mathbf{e}) \wedge \mathbf{s}(\mathbf{e}) = \mathbf{u} \wedge \mathbf{t}(\mathbf{e}) = \mathbf{v} \\ \wedge \gamma^* \wedge \text{Form}'(c', V \cup \{\mathbf{e}\}) \end{aligned}$$

Note that this exploits the correspondence between node identifiers and node variables established in the previous case.

For Boolean formulae over M-conditions, the transformation Form' is defined in the standard way, that is, $\text{Form}'(\neg c, V) = \neg \text{Form}'(c, V)$, $\text{Form}'(c \wedge d, V) = \text{Form}'(c, V) \wedge \text{Form}'(d, V)$, and $\text{Form}'(c \vee d, V) = \text{Form}'(c, V) \vee \text{Form}'(d, V)$. \square

To prove the theorem, we first prove a more general lemma about the translation of M-conditions.

Lemma 4 (M-conditions can be expressed as formulae). For every M-condition c in normal form, injective graph morphism $p: P \hookrightarrow G$, and interpretation I in G (defined for the free variables of c), we have:

$$p: P \hookrightarrow G \models^I c \text{ if and only if } I_G^{p, I} \models \text{Form}'(c, P^*)$$

where P^* is the set of node and edge variables corresponding to the identifiers in P . Furthermore, $I_G^{p, I}$ is defined as I_G but with the following mappings for free variables in φ : (1) for each set variable \mathbf{Y} in the domain of I , $I_G^{p, I}(\mathbf{Y}) = I(\mathbf{Y})$; (2) for each node v in X , $I_G^{p, I}(\mathbf{v}) = p(v)$; and (3) for each edge e in X , $I_G^{p, I}(\mathbf{e}) = p(e)$. \square

Proof. Cases $c = \exists_{\mathbf{V}} \mathbf{X}[c']$ and $c = \exists_{\mathbf{E}} \mathbf{X}[c']$ are clear from the definition of \models , the construction, and induction hypothesis. All other cases are easily adapted from the proof of Lemma 6.33 in [17]. \square

Proof (of Theorem 8). Define $i_G: \emptyset \hookrightarrow G$. We have that:

$$\begin{aligned} G \models c & \text{ iff } i_G \models^{I_\emptyset} c \\ & \text{ iff } I_G^{i_G, I_\emptyset} \models \text{Form}'(c, \{\}) \\ & \text{ iff } I_G \models \text{Form}'(c, \{\}) \\ & \text{ iff } G \models \text{Form}(c). \end{aligned}$$

from the definition of \models for M-conditions and formulae, and Lemma 4. \square

A.4 Proof of Theorem 1

Proof. We obtain the result directly from Theorems 7 and 8. \square

B Semantics of Graph Programs

This appendix contains an operational semantics – in the style of GP 2 [16] – for the graph programs defined in this paper. The semantics consists of inference rules, which inductively define a small-step transition relation \rightarrow on *configurations*. Intuitively, configurations represent the current state (a graph or special failure state) paired with a program that remains to be executed.

Definition 15 (Configuration). Let \mathcal{P} denote the class of all graph programs and \mathcal{G} the set of all graphs over \mathcal{C} . A *program configuration* is either a program with a graph in $\mathcal{P} \times \mathcal{G}$, just a graph in \mathcal{G} , or the special element fail. \square

Definition 16 (Transition relation). A *small-step transition relation*

$$\rightarrow \subseteq (\mathcal{P} \times \mathcal{G}) \times ((\mathcal{P} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\})$$

over configurations defines the individual steps of computation. The transitive and reflexive-transitive closures of \rightarrow are written \rightarrow^+ and \rightarrow^* respectively. \square

Configurations in $\mathcal{P} \times \mathcal{G}$ represent states of unfinished computations, whereas graphs in \mathcal{G} are proper results. The configuration fail represents a failure state. A configuration γ is said to be *terminal* if there is no configuration δ such that $\gamma \rightarrow \delta$.

We provide semantic inference rules for the commands of programs. Each inference rule has a premise and conclusion, separated by a horizontal bar. Both contain (implicitly) universally quantified meta-variables for programs and graphs, where \mathcal{R} stands for a rule set call, C, P, P', Q for programs in \mathcal{P} , and G, H for graphs in \mathcal{G} .

Definition 17 (Semantic inference rules for core commands). The *inference rules for core commands* of programs are given in Figure 7. The notation $G \not\Rightarrow_{\mathcal{R}} H$ expresses that for a graph G , there is no graph H such that $G \Rightarrow_{\mathcal{R}} H$. \square

To convey an intuition as to how the rules should be read, consider the rule $[\text{call}_1]_{\text{OS}}$. This reads: “for all sets of rules \mathcal{R} and all graphs G, H , $G \Rightarrow_{\mathcal{R}} H$ implies that $\langle \mathcal{R}, G \rangle \rightarrow H$ ”.

By inspection of the inference rules, we note that a program execution can only result in a failure state if a set of rules is applied to a graph for which no rule in the set is applicable.

The meaning of programs is given by the semantic function $\llbracket _ \rrbracket$, which assigns to each program P the function $\llbracket P \rrbracket$ mapping an input graph G to the set of all possible results of executing P on G . The application of function $\llbracket P \rrbracket$ to graph G is denoted $\llbracket P \rrbracket G$. As well as graphs, this set may contain the special values fail and \perp . The former indicates a program run ending in failure, whereas \perp indicates that at least one execution diverges (does not terminate), or “gets stuck”.

$$\begin{array}{c}
[\text{call}_1]_{\text{OS}} \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R}, G \rangle \rightarrow H} \quad [\text{call}_2]_{\text{OS}} \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R}, G \rangle \rightarrow \text{fail}} \\
[\text{seq}_1]_{\text{OS}} \frac{\langle P, G \rangle \rightarrow \langle P', H \rangle}{\langle P; Q, G \rangle \rightarrow \langle P'; Q, H \rangle} \quad [\text{seq}_2]_{\text{OS}} \frac{\langle P, G \rangle \rightarrow H}{\langle P; Q, G \rangle \rightarrow \langle Q, H \rangle} \\
[\text{seq}_3]_{\text{OS}} \frac{\langle P, G \rangle \rightarrow \text{fail}}{\langle P; Q, G \rangle \rightarrow \text{fail}} \\
[\text{if}_1]_{\text{OS}} \frac{\langle C, G \rangle \rightarrow^+ H}{\langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle P, G \rangle} \\
[\text{if}_2]_{\text{OS}} \frac{\langle C, G \rangle \rightarrow^+ \text{fail}}{\langle \text{if } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle Q, G \rangle} \\
[\text{try}_1]_{\text{OS}} \frac{\langle C, G \rangle \rightarrow^+ H}{\langle \text{try } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle P, H \rangle} \\
[\text{try}_2]_{\text{OS}} \frac{\langle C, G \rangle \rightarrow^+ \text{fail}}{\langle \text{try } C \text{ then } P \text{ else } Q, G \rangle \rightarrow \langle Q, G \rangle} \\
[\text{alap}_1]_{\text{OS}} \frac{\langle P, G \rangle \rightarrow^+ H}{\langle P!, G \rangle \rightarrow \langle P!, H \rangle} \quad [\text{alap}_2]_{\text{OS}} \frac{\langle P, G \rangle \rightarrow^+ \text{fail}}{\langle P!, G \rangle \rightarrow G}
\end{array}$$

Fig. 7. Inference rules for core commands

Definition 18 (Divergence). A program P *can diverge from* graph G if there is an infinite sequence:

$$\langle P, G \rangle \rightarrow \langle P_1, G_1 \rangle \rightarrow \langle P_2, G_2 \rangle \rightarrow \dots$$

□

Definition 19 (Getting stuck). A program P *can get stuck from* graph G if there is a terminal configuration $\langle Q, H \rangle$ such that $\langle P, G \rangle \rightarrow^* \langle Q, H \rangle$. □

A program can get stuck if the guard program C of a conditional can diverge on some graph G , neither producing a graph nor failing, or if the same property is true for a program that is iterated. The execution in these cases gets stuck because none of the inference rules for conditionals and iteration can be applied.

Definition 20 (Semantic function). The *semantic function* $\llbracket _ \rrbracket : \mathcal{P} \rightarrow (\mathcal{G} \rightarrow 2^{\mathcal{G} \cup \{\text{fail}, \perp\}})$, given a graph G and a program P , is defined by:

$$\begin{aligned}
\llbracket P \rrbracket G = & \{X \in \mathcal{G} \cup \{\text{fail}\} \mid \langle P, G \rangle \rightarrow^+ X\} \\
& \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}.
\end{aligned}$$

□

Finally, we provide a straightforward definition of program equivalence which is based on the definition of semantic functions.

Definition 21 (Semantic equivalence). Two graph programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $\llbracket P \rrbracket = \llbracket Q \rrbracket$. \square

C Weakest Liberal Precondition Constructions

C.1 Proof of Lemma 1

Proof. By structural induction.

Basis. Let $c = \text{true}$. Then $A'(p, \text{true}) = \text{true}$. All morphisms satisfy true.

Step. Only if. Let $c = \exists_v X[c']$. Assume that:

$$p'' \models^I A'(p, \exists_v X[c']) = \exists_v X[A'(p, c')].$$

Then there exists an interpretation $I' = I \cup \{X \mapsto V\}$ for some $V \subseteq V_H$ such that $p'' \models^{I'} A'(p, c')$. By induction hypothesis, we have $p'' \circ p \models^{I'} c'$. By definition of \models , we get the result that $p'' \circ p \models^I \exists_v X[c']$. Analogous for case $c = \exists_E X[c']$.

Let $c = \exists(a \mid \gamma, c')$. Assume that:

$$p'' \models^I A'(p, \exists(a \mid \gamma, c')) = \bigvee_{e \in \varepsilon} \exists(b \mid \gamma, A'(s, c'))$$

i.e. there exists an $e \in \varepsilon$ such that $p'' \models^I \exists(b \mid \gamma, A'(s, c'))$. By the definition of \models , there exists a morphism $q'' : E \hookrightarrow H$ with $p'' = q'' \circ b$, $\gamma^{I, q''} = \text{true}$, and $q'' \models^I A'(s, c')$. Directly from the proof of Lemma 3 in [7], we get $p'' \circ p \models^I \exists(a)$. Using the induction hypothesis, $q'' \models^I A'(s, c')$ implies $q'' \circ s \models^I c'$. Define $q' : C \hookrightarrow H$ as $q'' \circ s$. With the definition of \models , we have $p'' \circ p \models^I \exists(a, c')$. Finally, with the assumption that $\gamma^{I, q''} = \text{true}$, that γ is defined only for nodes and edges in C , and that $q' = q'' \circ s$, we have that $\gamma^{I, q'} = \text{true}$. Together, we have the result that $p'' \circ p \models^I \exists(a \mid \gamma, c') = c$.

Step. If. Let $c = \exists_v X[c']$. Assume that $p'' \circ p \models^I \exists_v X[c']$. Then there exists an interpretation $I' = I \cup \{X \mapsto V\}$ for some $V \subseteq V_H$ such that $p'' \circ p \models^{I'} c'$. By induction hypothesis, we have that $p'' \models^{I'} A'(p, c')$. By definition of \models and the construction of A' , we get the result that $p'' \models^I \exists_v X[A'(p, c')] = A'(p, \exists_v X[c'])$. Analogous for case $c = \exists_E X[c']$.

Let $c = \exists(a \mid \gamma, c')$. As for the “only if” direction, one can derive $p'' \models^I A'(p, \exists(a \mid \gamma, c'))$ directly from the proof of Lemma 3 in [7], with the additional requirement that $\gamma^{I, q'} = \gamma^{I, q''} = \text{true}$ for the satisfying morphisms $q' : C \hookrightarrow H$ and $q'' : E \hookrightarrow H$ (simple to show, because γ is defined only over items in C , and the proof in [7] shows that $q' = q'' \circ s$).

For Boolean formulae over M-conditions, the statement follows from the definition of \models and the induction hypothesis. \square

C.2 Proof of Proposition 1

Proof. Let $p = \text{path}(v, w, \text{not } E)$ with v, w denoting some nodes in V_R , and E a set of edges in E_R . Furthermore, given some morphism p , let $p(E)$ abbreviate the set $\{p(e) \mid e \in E\}$. We show that the equality holds for all contexts of v, w and all types of rules r (in the sense of what the rules create and/or delete).

Case (1). Suppose that $v, w \in V_K$ and $E_L = E_R$. Then $\text{LPath}(r, p)$ returns the predicate $\text{path}(v, w, \text{not } E)$. By the definition of interpretations, the equality holds if $[\text{path}_G(g(v), g(w), g(E)) \text{ iff } \text{path}_H(h(v), h(w), h(E))]$. If such a path exists in G then the same path exists in H (and vice versa), since the rule does not create or delete edges, and since any nodes created or deleted would not be part of such paths (by the dangling condition).

Case (2). Suppose that $v, w \in V_K$ and $E_R \subset E_L$. Then $\text{LPath}(r, p)$ returns the predicate $\text{path}(v, w, \text{not } E^\ominus)$. By the definition of interpretations, the equality holds if $[\text{path}_G(g(v), g(w), g(E^\ominus)) \text{ iff } \text{path}_H(h(v), h(w), h(E))]$. The argument is similar to that of the previous case, noting that edges r deletes are included in E^\ominus and hence are never part of such a path in G .

Case (3). Suppose that $v, w \in V_K$ and $E_L \subset E_R$. Then $\text{LPath}(r, p)$ returns:

$$\text{path}(v, w, \text{not } E) \text{ or } \text{FuturePaths}(r, p).$$

For paths along edges $e \in E_L, E_R$, the argument is as before. For paths including edges $e \in E_R \setminus E_L$, the equality then holds if:

$$[\text{FuturePaths}(r, p)^{I.g} \text{ iff } \text{path}_H(h(v), h(w), h(E))].$$

By construction, $\text{FuturePaths}(r, p)$ expands to a disjunction of:

$$(\text{path}(v, x_1, \text{not } E) \text{ and } \text{path}(y_1, x_2, \text{not } E) \dots \text{and } \text{path}(y_i, x_{i+1}, \text{not } E) \\ \dots \text{and } \text{path}(y_n, w, \text{not } E))$$

over all non-empty sequences of distinct pairs $\langle \langle x_1, y_1 \rangle, \dots, \langle x_n, y_n \rangle \rangle$ drawn from:

$$\{\langle x, y \rangle \mid x, y \in V_K \wedge \text{path}_R(x, y, E) \wedge \neg \text{path}_L(x, y, E)\}.$$

Assume that one such disjunct is satisfied in G . Then there are paths from $g(v)$ to $g(x_1)$, $\dots g(y_i)$ to $g(x_{i+1})$, \dots and $g(y_n)$ to $g(w)$ (all excluding edges in $g(E)$). These paths also exist in H , since each x, y pair is in the interface of r , since r does not delete edges, and since any nodes deleted would not have been part of the paths in G (to satisfy the dangling condition, all incident edges must also be deleted, which would lead to a contradiction). Furthermore, each endpoint $h(x)$ of these paths is connected to the beginning of the next one $h(y)$ (since $\text{path}_R(x, y, E)$, and each element of the path is injectively mapped to H). Together, we have a witness for $\text{path}_H(h(v), h(w), h(E))$.

Assume now that $\text{path}_H(h(v), h(w), h(E))$ holds by some path consisting of edges e_1, e_2, \dots, e_n in sequence, with $s(e_1) = h(v)$, and $t(e_n) = h(w)$. Let this

path be denoted by the pair $\langle h(v), h(w) \rangle$. Assume that at least one of its edges is in $h(E_R \setminus E_L)$, i.e. is created by r . The path contains path segments e_a, \dots, e_b in $h(E_R \setminus E_L)$, denoted by $\langle s(e_a), t(e_b) \rangle$, such that if there is an edge e_{a-1} (resp. e_{b+1}) in $\langle h(v), h(w) \rangle$, that edge is in the set $h(E_L)$. Since nodes created by r can only be incident to edges in $E_R \setminus E_L$ (by the dangling condition), there exists in G the same sequence of edges $\langle h(v), h(w) \rangle$ but with “gaps” for all such path segments $\langle s(e_a), t(e_b) \rangle$. Each pair of nodes $s(e_a), t(e_b)$ corresponds to a pair $x_i, y_i \in V_K$ such that $g(x_i) = h(x_i) = s(e_a)$ and $g(y_i) = h(y_i) = t(e_b)$. The construction returns a disjunct:

$$(\text{path}(v, x_1, \text{not } E) \text{ and } \text{path}(y_1, x_2, \text{not } E) \dots \text{and } \text{path}(y_m, w, \text{not } E))$$

for all such pairs x_i, y_i , since paths between them are created by r (that is, $\text{path}_R(x_i, y_i, E)$ holds and $\text{path}_L(x_i, y_i, E)$ does not). As the disjunct evaluates to true under I and g , so does $\text{FuturePaths}(r, p)$.

Case (4). Suppose that $v, w \in V_K$ and $E_L \neq E_R$, i.e. including the previous cases but also rules that both delete and create edges. Here, $\text{LPath}(r, p)$ returns:

$$\text{path}(v, w, \text{not } E^\ominus) \text{ or } \text{FuturePaths}(r, p)$$

with $\text{FuturePaths}(r, p)$ expanding as before but with E^\ominus replacing E . The argument is as in the previous case, but noting that edges r deletes are included in E^\ominus , and hence are not considered in H (nor in any corresponding path segments in G).

Case (5). Now, suppose that $v \notin V_K$, $w \in V_K$, and $E_L \neq E_R$ (the rule must create at least one edge from from v in V_R). Then $\text{LPath}(r, p)$ returns the constraint:

$$\begin{aligned} &\text{false or path}(x_1, w, \text{not } E^\ominus) \text{ or path}(x_2, w, \text{not } E^\ominus) \text{ or } \dots \\ &\dots \text{ or FuturePaths}(r, p) \end{aligned}$$

for each $x_i \in V_K$ such that $\text{path}_R(v, x_i, E^\ominus)$, and the equality holds if:

$$[\text{LPath}(r, p)^{I, g} = \text{true iff } \text{path}_H(h(v), h(w), h(E^\ominus))]$$

Assume that $\text{LPath}(r, p)^{I, g} = \text{true}$. Suppose that $\text{LPath}'(r, v, w, E^\ominus) = \text{true}$. Then from the construction, there exists some $z \in V_K$ with $\text{path}_R(v, z, E^\ominus)$ such that $(\text{path}(z, w, \text{not } E^\ominus))^{I, g} = \text{true} = \text{LPath}(r, \text{path}(z, w, \text{not } E^\ominus))^{I, g}$. By induction we get $\text{path}_H(h(z), h(w), h(E^\ominus))$. From the assumption $\text{path}_R(v, z, E^\ominus)$ and the definition of morphisms we derive that $\text{path}_H(h(v), h(z), E^\ominus)$, and hence the result that $\text{path}_H(h(v), h(w), h(E^\ominus))$. Suppose $\text{FuturePaths}(r, p)^{I, g} = \text{true}$ instead, i.e.

$$(\text{path}(z, x_1, \text{not } E^\ominus) \text{ and } \text{path}(y_1, x_2, \text{not } E^\ominus) \dots \text{and } \text{path}(y_m, w, \text{not } E^\ominus))$$

for some $z \in V_K$ such that $\text{path}_R(v, z, E^\ominus)$. Let $p' = \text{path}(z, w, \text{not } E^\ominus)$. Then:

$$\text{FuturePaths}(r, p)^{I, g} = \text{true} = \text{FuturePaths}(r, p')^{I, g} = \text{LPath}(r, p')^{I, g}.$$

By induction, we have that $\text{path}_H(h(z), h(w), h(E^\ominus))$. As before we derive that $\text{path}_H(h(v), h(z), E^\ominus)$, and hence the result that $\text{path}_H(h(v), h(w), h(E^\ominus))$.

Assume that $\text{path}_H(h(v), h(w), h(E^\ominus))$. There is a node $z \in V_K$ such that $\text{path}_R(v, z, E^\ominus)$ and $\text{path}_H(h(z), h(w), h(E^\ominus))$. (Suppose there is no such node. Then every node reachable from the image of v in H will also have been created by the rule, and in particular, none of these nodes will be the image of w since $w \in V_K$. A contradiction.) Let $p' = \text{path}(z, w, \text{not } E)$. By induction, we have that:

$$\text{LPath}'(r, p')^{I,g} = (\text{path}(z, w, \text{not } E^\ominus) \text{ or } \text{FuturePaths}(r, p'))^{I,g} = \text{true}.$$

If the first disjunct evaluates to true, then so does $\text{LPath}'(r, p)^{I,g}$ since the construction yields a disjunct $\text{path}(x_i, w, \text{not } E^\ominus)$ where $x_i = z$. If the second disjunct evaluates to true, i.e.

$$(\text{path}(z, x_1, \text{not } E^\ominus) \text{ and } \text{path}(y_1, x_2, \text{not } E^\ominus) \dots \text{and } \text{path}(y_m, w, \text{not } E^\ominus))$$

then so does $\text{LPath}'(r, p)^{I,g}$ since the above is yielded by the construction, i.e.

$$\text{LPath}'(r, v, x_1, E^\ominus) = \text{false or path}(z, x_1, \text{not } E^\ominus) \text{ or } \dots$$

Case (6). Suppose that $v \in V_K, w \notin V_K$, and $E_L \neq E_R$. Analogous to Case (5).

Case (7). Finally, suppose that $v, w \notin V_K$ and $E_L \neq E_R$. There are two subcases: (A) there are nodes in V_K which are reachable from v, w in R ; and (B) there are no such nodes in V_K . For the former subcase, the proof is along the lines of Cases (5)-(6). For the latter subcase, a path can only exist in H if there is a path from v to w in R . If there is, the construction returns the disjunct **true**; otherwise **false**. The equality clearly holds. \square

C.3 Proof of Lemma 2

Proof. By structural induction. Note that the construction distinguishes two cases in the step, according to whether a pushout complement exists or not. We will consider the case that one does; the other proceeds analogously to the proof of Theorem 6 in [7].

Basis. Let $c = \text{true}$. Then $L'(r, c, M) = \text{true}$. All morphisms satisfy true.

Step. Only if. Let $c = \exists_v \mathbf{X}[c']$. By assumption,

$$g \models^I L'(r, \exists_v \mathbf{X}[c'], M) = \exists_v \mathbf{X} \left[\bigvee_{M' \in 2^{M_v}} L'(r, c', M \cup M') \right].$$

There exists an M' such that $g \models^I \exists_v \mathbf{X}[L'(r, c', M \cup M')]$. By the definition of \models , there exists some $V \subseteq V_G$ such that $I' = I \cup \{\mathbf{X} \mapsto V\}$ and $g \models^{I'} L'(r, c', M \cup M')$. By induction, we get that $h \models^{I'_{M \cup M'}} c'$. Observe that:

$$I'_{M \cup M'} = I_M \cup \{\mathbf{X} \mapsto V \cup \{h(x) \mid (x, \mathbf{X}) \in M'\}\}.$$

By definition of \models , we get the result that $h \models^{I_M} \exists_v \mathbf{X}[c']$. (Analogous for case $c = \exists_e \mathbf{X}[c']$.)

Now let $c = \exists(a \mid \gamma, c')$. By assumption,

$$g \models^I L'(r, \exists(a \mid \gamma, c'), M) = \exists(b \mid \gamma_M, L'(r^*, c', M))$$

i.e. there is a morphism $q' : Y \hookrightarrow G$ such that $\gamma_M^{I, q'} = \text{true}$, $q' \circ b = g$, and $q' \models^I L'(r^*, c', M)$. Following the proof of Theorem 6 in [7], we derive a morphism $q : X \hookrightarrow H$ with $q \circ a = h$, i.e. $h \models^I \exists(a)$. Consider now the construction of γ_M from γ . Each MSO expression $x \in \mathbf{X}$ for $x \in X \setminus Y$ is replaced in γ_M by **true** (resp. **false**) if $(y, \mathbf{X}) \in M$ (resp. \notin) for some $y = x$. Observe that each $(x \in \mathbf{X})^{I_M, q}$ evaluates to the same Boolean value as the corresponding replacement (**true** or **false**) in γ_M under I, q' . Moreover, each path predicate p in γ is replaced with $\text{LPath}(r^*, p)$. By Proposition 1,

$$\text{LPath}(r^*, p)^{I, q'} = p^{I, q}.$$

Together, we have $\gamma_M^{I, q'} = \gamma^{I_M, q} = \text{true}$ and $h \models^{I_M} \exists(a \mid \gamma)$. From the induction hypothesis, $q' \models^I L'(r^*, c', M)$ implies $q \models^{I_M} c'$. With this, we get the result that $h \models^{I_M} \exists(a \mid \gamma, c')$.

Step. If. Let $c = \exists_v \mathbf{X}[c']$. By assumption, $h \models^{I_M} \exists_v \mathbf{X}[c']$. By the definition of \models , there exists some $V \subseteq V_H$ such that $h \models^{I_M \cup \{\mathbf{x} \mapsto V\}} c'$. Define:

$$M_{\mathbf{X}} = \{(x, \mathbf{X}) \mid x \in V_R \setminus V_L \wedge h(x) \in V\}$$

and $V^\ominus = V \setminus (V_H \setminus V_D)$. Now define:

$$I' = I \cup \{\mathbf{X} \mapsto V^\ominus\}$$

and hence:

$$I'_{M \cup M_X} = I_M \cup \{X \mapsto V^\ominus \cup \{h(x) \mid (x, X) \in M_X\}\}.$$

Observe that $h \models^{I'_{M \cup M_X}} c'$. By induction hypothesis, we get $g \models^{I'} L'(r, c', M \cup M_X)$. Clearly, M_X is in 2^{M_v} from the construction, hence:

$$g \models^{I'} \bigvee_{M' \in 2^{M_v}} L'(r, c', M \cup M').$$

By the definition of \models , we get the result that:

$$g \models^I \exists_v X [\bigvee_{M' \in 2^{M_v}} L'(r, c', M \cup M')] = L'(r, c, M).$$

(Analogous for case $c = \exists_E X [c']$.)

Now let $c = \exists(a \mid \gamma, c')$. By assumption, $h \models^{I_M} \exists(a \mid \gamma, c')$, i.e. there is a morphism $q: X \hookrightarrow H$ with $\gamma^{I_M, q} = \text{true}$, $q \circ a = h$, and $q \models^{I_M} c'$. Following the proof of Theorem 6 in [7], we derive a morphism $q': Y \hookrightarrow G$ with $q' \circ b = g$, i.e. $g \models^I \exists(b)$. Consider now the construction of γ_M from γ . Each MSO expression $x \in X$ for $x \in X \setminus Y$ is replaced in γ_M by **true** (resp. **false**) if $(y, X) \in M$ (resp. \notin) for some $y = x$. Observe that each $(x \in X)^{I_M, q}$ evaluates to the same Boolean value as the corresponding replacement (**true** or **false**) in γ_M under I, q' . Moreover, each path predicate p in γ is replaced with $\text{LPath}(r^*, p)$. By Proposition 1,

$$\text{LPath}(r^*, p)^{I, q'} = \text{LPath}(r^*, p)^{I_M, q'} = p^{I_M, q}.$$

Together, we have $\gamma_M^{I, q'} = \gamma^{I_M, q} = \text{true}$ and $g \models^I \exists(b \mid \gamma_M)$. From the induction hypothesis, $q \models^{I_M} c'$ implies $q' \models^I L'(r^*, c', M)$. Finally, we get the result that:

$$g \models^I \exists(b \mid \gamma_M, L'(r^*, c', M)) = L'(r, c, M).$$

For Boolean formulae over M-conditions, the statement follows from the definition of \models and the induction hypothesis. \square